EngageOne Generate

Production Guide

Version 6.6 Service Pack 11

Table of Contents

1 - Preface

2 - Working with Generate

3 - Running Generate in Server Mode

4 - Running Generate as a Started Task

5 - Programming PCE

6 - Running PCE

7 - Defining external keyed images

8 - Working with resources in a HIP file

9 - Processing PDF output

10 - Working with HTML

11 - User exits

12 - Structured XML journals

13 - Output datastream formats

14 - Appendix A - Generate SCP and Lookup Table codepage Overrides

1 - Preface

This section describes typographic and naming conventions used throughout this guide.

In this section

Conventions used in this guide

Typographic conventions

[...]	text between square brackets are optional.
{ opt1 | opt2 }	parameters between curly braces represent a list of options, one of which must be chosen.
Text in italics	represents parameter data which should be replaced with customized values.
UPPER CASE	text represents constant command text which should be typed exactly as written.
•	space character (used only if spaces are not apparent).

File naming conventions

The following conventions are expected whenever you need to specify file names in the Generate environment.

z/OS

All files are referenced by Data Definition (DD) labels with actual datasets being assigned to these labels in start-up JCL. Example:

```
Output=DD:AFPOUT
```

Windows

Files are referenced by path (optional) and filename. If a path name is not specified Generate will search the current directory (from which Generate was started) for the filename. Example:

```
Output=C:\DOC1HOST\AFPOUT\APPLIC1.AFP
```

UNIX

Files are referenced by path (optional) and filename. If a path name is not specified Generate will search the current directory (from which Generate was started) for the filename. Example:

```
Output=/doc1host/afpout/applic1.afp
```

Updates to this Guide This guide is issued in electronic format (PDF) only. It may be reissued from time to time to include corrections or additions that have been made since the original issue. These changes will be indicated with a change bar in the margins. The latest version of all product user guides can be downloaded from the DOC1 Support Net website.

2 - Working with Generate

Generate is the batch program that processes production jobs on your chosen host system. Generate reads information about the job requirements from a HIP file, merges the input data file it receives with your publication designs and produces output datastreams ready for printing or presenting on your intended output devices.

In this section

About Generate

The program typically has the name DOC1GEN on all platforms. You will need to run the program from the batch environment appropriate to your host system and production process: command line, script, JCL, etc.

The name and location of the HIP file that is to control a job is specified as a parameter to the DOC1GEN program when it is started. Normally, all other file references will have been specified in the production job settings that were used when the job was published in the Designer. If required however, you can create an Override Production Settings file (OPS) in which you can specify additional/alternative file references and other settings. Where used an OPS file is specified as a second parameter to the DOC1GEN start-up command.

> **Note:** On some supported platforms memory resident versions of DOC1GEN are available (Server Mode and Started Task). These allow a Generate production environment to remain loaded and for batches of input data passed to the defined channel to be processed dynamically. You will need to configure the environment before running in these modes and the launch method will differ from that described in this section. For more details see “Running Generate in Server Mode” and “Running Generate as a Started Task” in the Production Guide.

You will need to ensure that all input files referenced in the productions settings and the OPS file are available at the locations specified when DOC1GEN is started. For jobs running under z/OS indirect file references using DD names are typically used in which case the start up JCL will need to include DD cards with these labels that indicate the actual datasets.

Output files will be created with the file location/names specified in the production settings and the OPS file. You will need to ensure that suitable disk space is available to receive the output at the defined location. Under z/OS you will need to allocate suitable datasets either in advance or as part of start-up JCL if DD references are used.

Using segmented resources

Where you are using independently published (segmented) resources or Active Content you should be aware of the following:

- Generate needs to know where to find the segmented resources as they are not present in the design HIP file specified on the command line. The resource HIP or HIP's necessary for a job must be specified in the OPS file using the `<Input>` section `ResourceHIP` and `ActiveContentLocation` keywords. One or more `ResourceHIP` keywords can be used, allowing

device specific resources to be kept separate if desired. Segmented resources must be in the same repository as the publication design.
- Generate has no control over resource versions when working in this way. It is the user's responsibility to ensure that all HIPs are compatible and contain the appropriate resources.
- Care must be taken if resource HIPs specified by the `ResourceHIP` keyword contain conflicting device settings for the same device type. In this case the settings used by the last loaded HIP will be used, possibly resulting in unexpected output.
- The `<Output> Name` keyword is specified differently when resources are published independently. See **Output** in the OPS section.

Code page support

In order to read input data and configuration settings specified on your production system DOC1GEN needs to be able to convert the data it receives into the Unicode format it expects internally. To do this it uses a range of code page tables that define the required translations. For most Western applications these tables are contained within DOC1GEN itself and you need to take no specific action in the production environment. Due to their potentially large size, code page tables for non-Western applications are stored in a separate Extended Code Page (ECP) file and you will need to ensure this is made available to DOC1GEN by referencing it when starting the program.

Return codes

DOC1GEN always returns 0 (zero) for successful completion or where warning messages (only) have been issued. Return code 16 is issued where a failure has occurred – i.e. where an abort message has been issued.

Messages

Messages issued by Generate are always written to the standard output medium for the system on which the program is running. A list of possible messages and their explanations can be found on the EngageOne Compose Technical Support web site at **https://support.precisely.com/**.

Legacy support

The DOC1 Series 5 production engine cannot process jobs created in a pre-Series 5 environment without modification. However, you can use Generate to automatically launch the Series 4 production engine (known as EMFE). To do this you need to call the DOC1GEN program with an OPS file (see below for details) that indicates the location of the EMFE program and the relevant EMFE initialization file. You should refer to your Suite 4 user documentation for details of the parameters and files that are required when using this method.

OPS file

An override production settings file allows you to specify supplements and alternatives to some of the job settings that were used when publishing a publication design.

The use of an OPS file is optional unless you have not specified all file references and license data in the publishing task.

An OPS is a text file. Options are coded as keywords and associated parameters within several distinct sections. Sections must be introduced with the relevant name within angle brackets, for instance: `<Journal>`. If you want to include comments in the OPS file, prefix the comment line with a semicolon character.

No sections or keywords are compulsory and you should code only those options that suit your requirements. All missing options are assumed to have been specified as part of the publishing task.

It is important to note that doc1gen will not create folders it requires if they do not already exist. Any folders required by doc1gen must therefore be created before execution. All file references can include both path and file name as required. Ensure that you code all such references in a format suitable to the operating system under which you are running the production engine.

```
<Generate>
ProgramLocation=Filename
EMFE=Filename
INI=Filename
ServerMode={True|False}

<Input>
DataInput=Filename
MessageLib=Filename
MessagesFile=Filename
ResourceHIP=Filename
ActiveContentLocation=Path

<Journal>
Name =Filename
Name =Filename...

<LookupTable>
Name =Filename
Name =Filename
...

<LookupTableCodePages>
Name =CodePageValue
Name =CodePageValue
...
```

```
<KeyMap>
Name =Filename
Name =Filename
...
Mode ={Cached|Runtime}

<DIJ>
Name =Filename
...

<Output>
Name =Filename[,TempFilename]
Name =Filename[,TempFilename]
...
MessageAuditTrail=Filename

<eHTML>
BarcodeImageURL=URL
BarcodeFileTemplate=Filename
GraphicImageURL=URL
GraphicFileTemplate=Filename

<trace>
Outputfile=Filename
TraceLevel={off|default|verbose|complete|timing|completetiming}
outputcodepage={UTF8|default}
memlimit=Memory
publication=Number

<Messages>
MandatoryNotPlaced=Stop|Continue|Warn
MandatoryMessageError=Stop|Continue
OptionalMessageError=Stop|Continue
CampaignDate=String
Cycle=String
MessageProcessing={Yes|No}
NoMessages=Stop|Warn

<Server>
CommandQueue= {QueueID|PIPE:QueueID| SOCK:address:port|HOST:hostname:port}
Commandnn=CommandString
CommandBefore=CommandString
CommandOK=CommandString
CommandFail=CommandString
CommandEnd=CommandString AbortOnFail= {True|False}
...

<Advanced>
ErrorFile=Filename
LogFile=Filename
Checkpointfile=Filename
CPconsole= {0|1}
```

```
ConstantShapeOffPage=Abort|Warn|Ignore
DynamicShapeOffPage=Abort|Warn|Ignore
RangeOfPublications=n
WorkSpace=Filename
SystemTempFiles={Yes|No}
SuppressMessages={NONE|ALL|INFORMATION|<comma separated message IDs>}
ReportMemoryUsage={Yes|No}
eHTMLFluidReduceImagesToFit={Yes|No}
...

<OverFlow>
OverFLowFile=Filename
OverFlowSize=Memory

<Custom>
Name=Parameter
Name=Parameter
...
```

Sections, keywords and parameters:

<Generate>	If required, use this section to specify an alternative production engine program from that associated with the DOC1GEN program with which OPS was launched.
ProgramLocation	Can be used to indicate an alternative DOC1 Series 5 engine. Where used, Filename must indicate a PCOM.DLL file (or equivalent name depending on your production platform) as supplied with versions of Generate.
EMFE	Can be used to indicate a DOC1 Suite 4 production engine. Where used, Filename must indicate an EMFE.EXE file (or equivalent name depending on your production platform). If you use this option you must also code the INI keyword to specify an EMFE initialization file that contains the settings for the job to be launched. If you use the EMFE keyword all other OPS settings are ignored.
ServerMode	Setting the keyword to True initiates Server Mode. Setting the keyword to False overrides any Server Mode settings contained in the HIP file and DOC1GEN runs in batch mode.
<Input>	

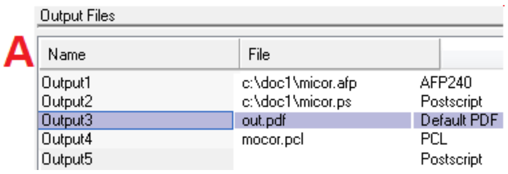
DataInput	Filename specifies the main Designer input file; i.e. the keyed record, delimited or XML file that will provide variable data to the current job.
MessageLib	Filename specifies the DOC1GEN production message library (PCOMEng.DLL or equivalent). Use this option if the message library file is not in the same location as the DOC1GEN program with which the OPS is to be launched. Also use it if the ProgramLocation or EMFE keywords are being used to specify an alternative production engine in which case the message file associated with the selected engine should be specified.
MessagesFile	Filename specifies the Message1/ Content Author HIM files to be used (note that you can use wildcard characters in the filename, e.g. re*.him). These files contain the required messages.
ResourceHIP	Filename specifies the resources file used when publishing output definition separately. Each additional HIP specified relates to a resources HIP targeted at a specific device. These must pair up with entries in the <Output> section.
ActiveContentLocation	Used when publishing Active Content separately. Path specifies the location of the Active Content files. Use %1 in the path as a place holder for the Active Content HIP file name. This name is generated automatically by Designer when publishing the Active Content and must not be changed. It is in the form Rxxxxxxx, where **x** is an alphanumeric character, for example `R000012A`**.** If %1 is omitted, the Active Content file name (Rxxxxxxx) will be appended to Path. This is an acceptable format for Windows and UNIX, but care should be taken on mainframe systems such as z/OS.
<Journal>	This section allows you to specify or override the file references to be used when creating journal files for the publication. Name should be the file alias assigned to a journal object in the publication design. Filename should indicate the actual file to receive the journal output. You may code as many entries as necessary to meet the number of journals to be created by the publication.

<LookupTable>	This section allows you to specify or override the file references to be used with lookup table functions within the publication. Name should be the file alias assigned to a lookup table function in the publication design. Filename should indicate the actual file containing the lookup table data. You may code as many entries as necessary to meet the number of lookup tables referenced by the publication.
<LookupTableCodePages>	This section allows you to associate code page values to lookup table assignments defined in the <LookupTable> section. Refer to **Generate SCP and lookup table override values** on page 352 for details on lookup table code page override values.
<KeyMap>	This section allows you to specify or override the settings used with key maps within the **External Key Maps** section of a production job or Publish Wizard.
Name	The file alias assigned to a key map in the publication design. Filename should indicate the actual file containing the key map data. You may code as many entries as necessary to meet the number of key maps referenced by the publication.
Mode	Specifies how the key map is read at run time and is applied to all the key maps associated with a publication. For more details, see the Publishing and Deployment section in the Designer User's Guide.
	Cached will copy the key map into memory, making access much faster. This is the default setting. **Runtime** will access the key map externally each time. This does not use any extra memory, but will not be as fast. You may want to use this option if there are a great number of entries in the map and memory is limited.
<DIJ>	This section allows you to override the settings specified for the Document Interchange Journal in the **Output Files** section of the Publish Wizard. See Appendix C in the Designer User's Guide for details. **Name** corresponds to the **Name** field which is automatically assigned when using the Publish Wizard. **Filename** corresponds to the **Document Interchange Journal** field and indicates the actual DIJ file you want to use.

<Output>

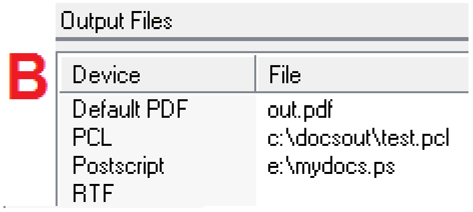
This section allows you to specify or override the file references to be used when creating the output datastream files produced by the job.

Name



Output Files

Name	File	
Output1	c:\doc1\micor.afp	AFP240
Output2	c:\doc1\micor.ps	Postscript
Output3	out.pdf	Default PDF
Output4	mocor.pcl	PCL
Output5		Postscript



Output Files

Device	File
Default PDF	out.pdf
PCL	c:\docsout\test.pcl
Postscript	e:\mydocs.ps
RTF	

A - Use the **Name** for regular publishing

B- or the **Device** name when publishing the resources separately.

The **Name** is the reference name assigned to the output file when publishing the publication and optionally, a temporary file.

Filename indicates the actual file containing the datastream. If set to 'null' the publication will be not be generated for that output device, e.g. output2=null.

If required, you can override the name of the temporary file used by Generate for a specific output device using ***TempFilename***. This is used in preference to the **Workspace**. When specifying output datastream files you may code as many entries as necessary to meet the number of output datastreams to be created by the publication.

MessageAuditTrail

Filename is the file to be used for Message1 and Content Author audit information for each message that is included in the published documents.

If you specify %d in the file name, for example:

```
MessageAuditTrail=trace%d.out.txt
```

then %d in the file name will be replaced with the date and time the file was generated.

<eHTML>

BarcodeImageURL	URL is the location where barcode images will be placed for embedding in the HTML pages.
BarcodeFileTemplate	Filename is the base file name to be used for barcode GIF images. It must be in a format suitable for the host platform. Two parameters can be included to make the filename unique – %1 is the generated filename and %2 is an index number.
GraphicImageURL	URL is the location where graphics images will be placed for embedding in the HTML pages.
GraphicFileTemplate	Filename is the base file name to be used for graphics JPG images. It must be in a format suitable for the host platform. Two parameters can be included to make the filename unique – %1 is the generated filename and %2 is an index number.
<Trace>	
OutputFile	Filename is the name of the file for the trace information. If no file is specified the output is sent to the standard output medium of the Generate host environment (e.g. command prompt window, system log, etc.). If you specify %d in the file name, for example: `OutputFile=traceouput%d.txt` then %d in the file name will be replaced with the date and time the file was generated.
TraceLevel	This controls the amount of trace information that is output. **Off** writes nothing i.e. turns trace off. **Default** shows the path that Generate took when the error occurred. This is in the form of a tree structure. This enables the error to be tracked down to a particular object in the publication design. Verbose and Complete modes include additional information such as internal references and the instructions that are being executed. These settings are used by Precisely Support to help troubleshoot when necessary. The Timing modes adds additional information to the trace on the relative time taken to execute parts of the logic. The times are measured in ticks, which are machine-dependent units of time.

OutputCodePage	This allows you to change the code page for the output file from the **default** host code page to the general Unicode **UTF8** code page.
MemLimit	Controls the maximum amount of memory that the trace module can use for buffering trace information. A value of '0' means no limit.
Publication	Allows the user to specify the **number** of a publication to be traced in addition to the first publication that causes the error. This will only be traced if it occurs before the publication with the error (that causes the trace to stop processing).
<Messages>	
MandatoryNotPlaced	Select the action you want Generate to take when a mandatory message cannot be included in a document for which it was intended. The default is to **Stop** processing, otherwise you can ignore the error and **Continue** or issue a **Warning** and continue.
MandatoryMessageError	Select the action you want Generate to take when a mandatory message has unresolved links – typically when a font used by the message is not included in the resource pack or when a data field used has not been mapped. The default is to **Stop** processing or you can ignore the error and **Continue** processing.
OptionalMessageError	As above but for non-mandatory messages.
CampaignDate	Specifies the date to be used when selecting messages using the activation and expiration attributes as defined within the Message1 and Content Author environments. Options are: **Auto** – the current system date **Auto+**|-<n><d|w|m> – the current system date plus or minus the specified number of days, weeks or months, e.g Auto+10d, Auto-3w **dd/mm/yyyy** or **mm/dd/yyyy** – a specific date
Cycle	String defines which cycle (defined in the Message1 and Content Author environments) to use. If defined, only messages belonging to this cycle will be selected.

MessageProcessing	Allows details for message rejections to be output to the standard output medium for the system on which the program is running. The default is **No**.
NoMessages	Specify the Generate action to be taken if messages are expected, but none can be found in the Messages file. The default is to **Stop** processing, or you can ignore the error, issue a **Warning** and continue.
FontMappingFromHip	Setting this to **Yes** will ensure that any mappings that have been applied to fonts used in a publication in the Designer (i.e. are in the HIP file) will also be applied to the same fonts used in messages created in Content Author or Message1. The default is **No**.
<Server>	The keywords in this section are used with the Server Mode method of running DOC1GEN under UNIX and Windows. For more details on Server Mode, refer to **Running Generate in Server Mode** on page 33 .
CommandQueue	Including this keyword in the OPS file initiates server mode. **QueueID** – is the name of the communication channel in the format applicable to the operating system. Under UNIX this is the name of the existing pipe and under Windows this is the name of a pipe which will be set up automatically when Server mode is initiated. This must always use the following conventions: `\\.\pipe\name` **address:port** – is the name of a communication channel specified as a TCP/IP 'dot' address and an associated port number, such as `10.133.54.202:5000` **hostname:port** – is the name of a communication channel specified as a host name known to the current system and an associated number, such as `spa02:5000`
Commandnn	– The number **nn** is used to reference the command when the DOC1SBMT program is executed. The **command string** can pass up to nine parameters which can be user defined or predefined elements of the Server Mode environment. The predefined elements are identified by fixed symbols as in the following list (assumes that defaults are being used, i.e. a **ValuePrefix** of “&” and **ValueSuffix** of “.”): **&I.** – input file name **&Pn.** – print file name (where **n** is the file alias assigned to the output file when publishing a publication, i.e. &POutput1. &POutput2.)

	&R. – server mode return code… 0 job OK 10 PAGE 1 of N overflow threshold exceeded 15 error executing system command specified by the user 25 failed during processing of job 30 failed during termination of job 50 failed during initialization of processing. Most likely input or print filenames were invalid
	&S. – job submission comment generated automatically by Server Mode to identify the process uniquely **&Jn.** – journal file (where **n** is the file alias assigned to a journal object, e.g. &JDOCJ1. &JDOCJ2.) **&Dn.** – Document Interchange Journal (where **n** is the file alias assigned to a DIJ object, i.e. &Ddij1.) **&C.** – pages generated to the point at which the command is called
CommandBefore	Optionally identifies a system command that will be executed prior to the start of processing application data with DOC1GEN. See **Commandnn** for more information.
CommandOK	Optionally identifies a system command to be executed only after the application data has been processed successfully by DOC1GEN. See **Commandnn** for more information.
CommandFail	Optionally identifies a system command to be executed only when the application data fails to be processed successfully by DOC1GEN. See **Commandnn** for more information.
CommandEnd	Optionally identifies a system command to be executed only after the application data has been processed successfully by DOC1GEN. See **Commandnn** for more information.
AbortOnFail	This defines the behavior of DOC1GEN if a print job run in Server Mode fails. The default **True** will abort Server Mode, while **False** gets Server Mode to attempt to discard the failing print job and await further commands.
<Advanced>	

ErrorFile	This option is used to specify the file that will receive any publication datasets that cannot be processed by Generate when a production job is run. It will override the file specified in the **Data record file** option in the Publish Wizard.The **filename** must be in the required format for the operating system.
LogFile	This option is used to specify the file that will receive any error or warning messages issued by Generate. The **filename** must be in the required format for the operating system.
Checkpointfile	This option is used to specify the file that will receive the messages that indicate which publication data set is currently being processed.
CPconsole	Checkpointing messages can be reported to the standard output medium of the Generate host environment (e.g. command prompt window, system log). Set to 0 – do not report messages (default) 1 – switch reporting on.
#Restart	When this is included after Generate has failed, it will restart and continue processing from the last checkpoint using the information in the checkpoint file. See also the Publish Wizard checkpoint progress option in the Designer User’s Guide. This can be included anywhere in the OPS file.
RangeOfPublications	This is used if you want Generate to process only a subset of the publication data sets available in the input data file. You may want to do this if you need to rerun portions of a production job without creating a new input data file. You can indicate the sequential numbers of the publication data sets to be processed as follows: `27,280,674` – specific publications `100-1000` – publications between 100 and 1000 `1000+` – all publications after the first 1000 `366,500-1000,2000+` – combinations.

ConstantShapeOffPage	Defines the action Generate should take if graphic objects (including text boxes) positioned using constant values for both X and Y offsets are positioned all or in part outside the active logical page area. Options are: • ***Abort*** – Generate aborts immediately. Any output files that have been created by the job are deleted (if this is permitted by the host operating system). • ***Warn*** – a warning message is issued for each object that is found to be positioned outside the logical page area. Processing of the job continues as normal. The off page object is included in the output datastream; the effect of this in the printer/browser environment will depend on the device type. • ***Ignore*** – as above but no warning message is issued. This is the default for objects placed using constant values.
DynamicShapeOffPage	As above but this option applies to graphic objects positioned using variable data for either offset.
WorkSpace	This option specifies a file template used by Generate to create temporary files at runtime. Refer to the section on creating a host object in the Designer User's Guide for further information Use either the %1 or %2 placeholders to create unique filenames, refer to the Publish Wizard checkpoint progress option in the Designer User's Guide for further information. This option is not for use on z/OS. However you can define a temporary file explicitly when specifying the output file, see **output file** for details.
SystemTempfiles	When set to **Yes** the host operating system will allocated temporary files for Generate to use at runtime. Note that either this option or the Workspace option should be used to manage temporary files.
SuppressMessages	Use this setting to indicate the level message suppression used by Generate for your production job. You can either indicate the category of messages to be suppressed or indicate specific messages that you do not want to be reported. Choose from one of the options that follows: • **NONE** - No messages suppressed • **INFORMATION** - All information messages suppressed • **ALL** - All warning and information messages suppressed • A list of comma separated message IDs to suppress (e.g. 121,420) When using this option you must specify only the identifier that appears before each message when issued.

ReportMemoryUsage	This option causes Generate to produce a message reporting the total amount of memory in bytes allocated during the production run. The message is reported on the command line and in any logfile specified. Valid options are Yes and No; note that the default is No.
eHTMLFluidReduceImagesToFit	When set to **Yes**, this setting allows images in Generate HTML for e-mail (eHTML) output to be sized in accordance with the resizing of the e-mail client.
<Overflow>	These options are used to override the **Limit composed pages in memory** settings on the **Memory Handling** page in the Publish Wizard. For details, see the section on error handling in the Publish Wizard options of the Designer User's Guide.
OverflowEnabled	When set to **No** – the default value – the **OverflowFile** and **OverflowSize** settings only take effect if the **Limit composed pages in memory** option is specified in the Publish Wizard. When set to **Yes**, the **OverflowFile** and **OverflowSize** settings always take effect.
OverFlowFile	This option designates the temporary file to which the composed files are written.
OverFlowSize	This option specifies the memory limit at which the process of writing to the overflow file begins. The default memory value is 4 megabytes. You can use the suffix K to indicate kilobytes, M to indicate megabytes or if no suffix is used the value will be in bytes.
<Custom>	This section allows you to specify any temporary settings that may be required as part of problem resolution. The keyword Name and associated Parameter will be provided directly by Precisely Support as required. You may code as many entries in this section as necessary.
The following Custom overrides relate to Input file caching and may be used to resolve memory handling issues where appropriate.	
DataFieldBufferFile	This option designates the temporary file to which data fields are written.

DataTextBufferFile	This option designates the temporary file to which text data from data fields is written.
DataBufferThreshold	This option determines how much memory is used by data fields before caching to disk.
ReadAheadThreshold	This option determines how much of the disk cache is read in to memory per read.

Note: that `DataBufferThreshold` and `ReadAheadThreshold` assignments default to byte values. Alternatively, you can use “K” (for Kilobytes) and “M” for “Megabytes” e.g 100K , 10M, etc. Refer to the example that follows for further details.

Example

```
<Generate>
ProgramLocation=\\servnet\mt\doc1gen.exe

<Input>
MessageLib=\\servnet\mt\doc1msg.dll
ResourceHIP=C:\Resources\Default pdf.hip
ResourceHIP=C:\Resources\myafp240.hip
ActiveContentLocation=C:\Resources\%1

<Journal>
J1=\trace\docj1.txt
J1=\trace\docj2.txt
J1=\trace\docj3.txt

<LookupTable>
tsub=\\servnet\mt\lookups.txt
tsub=\\servnet\mt\lookups1.txt

<LookupTableCodePages>
tsub=UTF8
tsub1=iso-8859-1

<KeyMap>
kmap1=\\resnet\gjk\km1.xml mode=Runtime

<DIJ>
Output1=\doc\edm\doca.jrn

<Output>
Output1=eas21.afp
```

```
Output2=eas21.pdf,temp.pdf
Output3=C:\PDF\test1.pdf
Output4=C:\AFP\test1.afp

<ehtml>
BarcodeImageURL=http://doc/html/resources/
BarcodeFileTemplate=Bar%1-%2.gif
GraphicImageURL=http://doc/html/resources/
GraphicFileTemplate=Graphic%1-%2.jpg

<Trace>
Outputfile=trace.out
TraceLevel=default output
codepage=utf8
memlimit=0
publication=3

<Messages>
MandatoryNotPlaced=Warn
MandatoryMessageError=Continue
OptionalMessageError=Continue
CampaignDate=12/07/2007
Cycle=AC02
MessageProcessing=Yes
NoMessages=Warn

<Server>
CommandQueue=HOST:spa02:5001
Command00=¨sndmsg msg('DOC1GEN processed &I.') tousr(*requester)¨
Command01=¨sndmsg msg('&0.') tousr(*requester)¨
CommandBefore=¨time /t >> start.log¨
CommandOK=¨time /t >> OK.log¨
CommandFail=¨time /t >> Fail.log¨
CommandEnd=¨time /t >> End.log¨
AbortOnFail=False

<Advanced>
ErrorFile=doc\backups\june21err.txt
LogFile=trace04.out
Checkpointfile=check.out
CPconsole=1
RangeOfPublications=100-350
ConstantShapeOffPage=Warn
DynamicShapeOffPage=Warn
WorkSpace=d:\process\work\b%1xml
SystemTempfiles=yes
ReportMemoryUsage=Yes
eHTMLFluidReduceImagesToFit=Yes

#restart
<OverFlow>
OverFlowFile=doc\memerror.txt
OverFlowSize=48m
```

```
<Custom>
PTF5690=¨Type1¨
DataFieldBufferFile=path\DataFieldBuffer_filename.buf
DataTextBufferFile =path\DataTextBuffer_filename.buf
DataBufferThreshold=10M
ReadAheadThreshold=100K
```

Using symbols

You can dynamically define parameters used in an OPS file by using symbols when starting a Generate job. The value assigned to a symbol is substituted wherever it is referenced in the OPS file and can be used to provide part or all of any parameter.

Where used, symbols must be defined after any other parameters in the start-up syntax. Under Windows for example, you could specify the following on the command line:

```
doc1gen j1.hip ops=j1.ops ext=txt
```

where ext is a symbol name. When referenced in the OPS file the symbol names must be enclosed in percent (%) characters. For example:

```
<Journal>
J1=\trace\docj1.%ext%
J1=\trace\docj1.%ext%
...
```

Symbol names are case sensitive. Where a symbol is referenced in the OPS but no value is assigned it is treated as an empty string.

Note that the following must not be used as symbol names: ops, mode, ecp, mmgx

Symbols may also be defined within the OPS file itself by coding them in an <OPS> section prior to where they need to be referenced. The format is as follows:

```
<Symbols>
Name =Parameter
Name =Parameter
...
```

If the same symbol name is specified both in start-up parameters and in the OPS itself then the start-up parameter will override. For example; in the OPS:

```
<Symbols>
RunName=run
BaseDir=\doc

<Journal>
J1=%BaseDir%\trace\docj1.txt
J2=%BaseDir%\trace\docj2.txt
J3=%BaseDir%\trace\%RunName%j3.txt

<Output>
Output1=%RunName%1.afp
Output2=%RunName%1.pdf
```

and on the Windows command line:

```
doc1gen in.hip OPS=OFile RunName=tst
```

the ‘J3’ journal name will be specified as `\trace\tst.txt`.

Running Generate under z/OS

Preparation:

A DOC1GEN job is typically submitted to the system via standard JCL.

The JOBLIB and STEPLIB concatenation must reference the main Generate load library and message library datasets. You may also need to include references to the IBM Language Environment (LE) run-time libraries if these are not known to system libraries.

The HIP file that controls the job plus an OPS file (where used) are specified as parameters on the EXEC card. However, these normally indicate DD references that are resolved to dataset names in subsequent DD cards.

The JCL must also include DD cards for all other files that have been identified using DD references in the HIP or OPS.

No two output files should be members of the same dataset.

EXEC card syntax:

```
EXEC PGM=DOC1GEN,PARM=('DD:HipRef [,OPS=DD:OpsRef ] [,ECP=DD:EcpRef ]
[,SCP=CodePage ] [,#restart] [,symbols]')
```

Parameters:

HipRef

is the DD label indicating the HIP file that will control the job.

OpsRef

is the DD label indicating an override production settings file if appropriate.

EcpRef

is the DD label indicating the Extended Code Page file which will be required for most non-Western applications. The ecp file must be placed in a fixed block dataset.

CodePage

is the number of a host code page to be used instead of the default code page – US (37). Refer to **Generate SCP and lookup table override values** on page 352 for details on host code page numbers

#restart	restart the job from the last checkpoint position. See the Publish Wizard checkpoint progress option in the Designer User’s Guide.
symbols	See **Using symbols** on page 27.

Example JCL

```
//DOCJOB6 JOB  '5438','JDOE',CLASS=F,REGION=2M
//DOC1GEN  EXEC PGM=DOC1GEN,
//      PARM='DD:DOCHIP,OPS=DD:DOCOPS,ECP=DD:DOC1ECP'
//*Generate load libraries. You may need to add run-time libs
//STEPLIB  DD DISP=SHR,DSN=PROD.DOC.LOAD
//         DD DISP=SHR,DSN=PROD.DOC.MSGS
//*HIP & OPS files
//DOCHIP  DD DISP=SHR,DSN=PROD.DOC.RUN(JOB6HIP)
//DOCOPS  DD DISP=SHR,DSN=PROD.DOC.RUN(NEWFILES)
//*Extended Code Page file
//DOC1ECP  DD DISP=SHR,DSN=PROD.DOC.RUN(DOC1ECP)//*Input data (as per
DD ref in HIP or OPS)
//DOCINPT DD DISP=SHR,DSN=PROD.DOC.DATA
//*Lookup tables (as per DD refs in HIP or OPS)
//DOCTL1  DD DISP=SHR,DSN=PROD.DOC.RUN(JOB6TL1)
//DOCTL2  DD DISP=SHR,DSN=PROD.DOC.RUN(JOB6TL2)
//*Output datastreams (as per DD refs in HIP or OPS)
//AFPOUT1  DD SYSOUT=X,DCB=LRECL=8205
//AFPOUT2  DD SYSOUT=X,DCB=LRECL=8205
//*Journals (as per DD refs in HIP or OPS)
//DOCJRN1 DD DISP=SHR,DSN=PROD.DOC.RUN(JOB6JRN1)
//DOCJRN2 DD DISP=SHR,DSN=PROD.DOC.RUN(JOB6JRN2)
```

Running Generate under UNIX and Windows

Preparation:

DOC1GEN is executed from the command prompt.

The HIP file that controls the job plus an OPS file (where used) are identified as parameters to the start up command.

All other references to files to be used or created by Generate are defined within the HIP or OPS files. You will need to ensure that these are available (or creatable) at the locations indicated.

Syntax:

```
doc1gen HipRef [ops=OpsRef] [ecp=EcpRef] [SCP=Codepage] ”#restart”
[symbols]
```

Parameters:

HipRef

is the path/file name of the HIP file that will control the job.

OpsRef

if an override production settings file is being used this is the path/file name of the OPS.

EcpRef

is the path/file name of the Extended Code Page file which will be required for most non-Western applications.

Codepage

is the number of a host code page to be used as an override for the Application Data code page. Refer to **Generate SCP and lookup table override values** on page 352 for details on host code page numbers

#restart

restart the job from the last checkpoint position. See the Publish Wizard checkpoint progress option in the Designer User’s Guide. Note that the quotes are only required when running under UNIX.

symbols

See **Using symbols** on page 27.

Examples

For Windows:

```
C:\doc\run\doc1gen job6.hip ops=C:\doctemp\newfiles.ops ecp=doc1ecp
```

For UNIX:

```
/doc/run/doc1gen job6.hip ops=C/doctemp/newfiles.ops ecp=doc1ecp
```

3 - Running Generate in Server Mode

Server Mode is a method of running DOC1GEN under UNIX and Windows so that the program responds to appropriate commands to process application data and, optionally, to execute associated system commands.

Such associated commands can be specified to be executed before Server Mode has processed a batch of application data, if it completes successfully or if it fails to complete successfully. They can be any commands that you can pass to the appropriate operating system from the command line. This allows a great deal of flexibility as you can use them, for instance, to log information, send the output to a network location or delete files after printing.

In this section

Server Mode Environment

The Server Mode environment is configured by settings in the <Server> section of an Override Production Settings (OPS) file used when launching DOC1GEN, for details see **Override production settings (OPS) File**. Once initiated, it will then remain active waiting for application data to be submitted to it for processing.

> **Note:** For a detailed summary on running DOC1GEN and using OPS files, see “Running Generate” in the Designer User’s Guide.

You can execute multiple instances of DOC1GEN Server Mode at the same time with each instance having a different environment loaded according to the OPS file used.

Server Mode executes the standard DOC1GEN program in a memory-resident format. You do not require a different version of the DOC1GEN program to be able to use it in Server Mode.

Command Queue

Server mode uses a command queue to receive commands and application data. These are more commonly known as "pipes" under UNIX and Windows. The command queue can be specified either as a named pipe or a TCP/IP socket.

Where used, the TCP/IP socket must specify a port number that is not already being used on the target system. Generally speaking, a port number <6000 will be unused but if you are unsure you should contact your system administrator.

When using a TCP/IP socket or a pipe under Windows the queue is set up and removed automatically by Server mode.

When using a named queue under UNIX it must be set up in advance by the user and must always be specified as FIFO (first in, first out) format. The mknod utility is used to initiate a UNIX pipe. Details of this utility follow later in this section.

Note that if you are running more than one instance of DOC1GEN in Server Mode then you must set up a different queue for each instance.

Control Programs

To use DOC1GEN in Server Mode two additional programs are provided with Generate Host distribution material:

DOC1SBMT	Submits application data to a Server Mode process
DOC1QUIT	Terminates a Server Mode process

Details of these programs follow later in this section.

To change the DOC1GEN environment used by a particular Server Mode process you will have to DOC1QUIT and then restart DOC1GEN in Server Mode with new parameters.

Specifying the command queue

The command queue is specified in the **CommandQueue** keyword of the appropriate OPS file.

CommandQueue has three sub-keywords – PIPE:, SOCK: or HOST: – that indicate the queue method being used.

PIPE: indicates the use of a named queue. It is assumed by default and the sub-keyword can be omitted. Under NT pipe names used a fixed format and must be specified exactly as indicated in the examples below, i.e. only the actual queue name (**cmdqueue** in the examples) can be different. Under UNIX any valid path/file name can be used.

SOCK: indicates a TCP/IP socket identified by IP address and port number.

HOST: indicates a TCP/IP socket identified by host name and port number.

The following examples indicate the available methods for specifying the queue:

Named pipe under NT

```
CommandQueue=\\.\pipe\cmdqueue
```

Named pipe under UNIX

```
CommandQueue=/temp/cmdqueue
```

Optionally use PIPE: form for named q's

```
CommandQueue=PIPE:/temp/cmdqueue
```

Use SOCK: form for IP address and port

```
CommandQueue=SOCK:10.133.54.202:6000
```

Use HOST: form for host name and port

```
CommandQueue=HOST:spa02:6000
```

Memory Allocation

When running in Server Mode you may not be sure of the size of the application data – and therefore the size of system memory requirements – that is to be submitted to the Server Mode process. If this is a concern you can use the memory settings in the publication wizard.

By default Server Mode allocates memory as required. Using the **Memory Allocation** settings in the publish wizard you can choose to customize options, such as how much memory to allocate and whether you want to abort a job if the amount of memory exceeds a predefined limit. For more information see “Running a publishing task” in the Designer User’s Guide.

Specifying System Commands

System commands to be performed in association with the process are predefined in the Server section of the appropriate OPS file and then referenced by DOC1SBMT program via the number assigned to them. You can dynamically pass parameters to be used with these commands.

Override production settings (OPS) File

Note that Server Mode gets the parameters for input file, print file, and journal list as part of DOC1SBMT rather than from the OPS file as with a standard DOC1GEN process. If any these parameters are specified in the OPS file they are ignored.

OPS file format

Syntax:

```
<Server>
CommandQueue= {QueueID|PIPE:QueueID|SOCK:address:port|HOST:hostname:port}
Commandnn=CommandString
CommandBefore=CommandString
CommandOK=CommandString
CommandFail=CommandString
CommandEnd=CommandString
AbortOnFail= {True|False}
...
```

Keywords and parameters: The following details provide information on the keywords used in the server section of the OPS file.

CommandQueue

including this keyword in the OPS file initiates server mode.

QueueID – is the name of the communication channel in the format applicable to the operating system. Under UNIX this is the name of the existing pipe and under Windows this is the name of a pipe which will be set up automatically when Server mode is initiated. This must always use the following conventions: `\\.\pipe\name`

address:port – is the name of a communication channel specified as a TCP/IP 'dot' address and an associated port number, such as `10.133.54.202:6000`

hostname:port – is the name of a communication channel specified as a host name known to the current system and an associated number, such as `spa02:6000`

> **Note:** Note that the ‘&’ character is a reserved character and must not be used when specifying paths or names.

OPS file format

Commandnn

the number nn is a two digit number between 00-99 used to reference the command when the DOC1SBMT program is executed. The **command string** can pass up to nine parameters which can be user defined or predefined elements of the Server Mode environment. The predefined elements are identified by fixed symbols as in the following list (assumes that defaults are being used, i.e. a **ValuePrefix** of "&" and **ValueSuffix** of "."):

&I. – input file name

&Pn. – print file name (where **n** is the file alias assigned to the output file when publishing the publication, i.e. &POutput1. &POutput2.)

&R. – server mode return code…

- 0 job OK
- 10 PAGE 1 of N overflow threshold exceeded
- 15 error executing system command specified by the user
- 25 failed during processing of job
- 30 failed during termination of job
- 50 failed during initialization of processing. Most likely input or print filenames were invalid

&S. – job submission comment generated automatically by Server Mode to identify the process uniquely

&Jn. – journal file (where **n** is the file alias assigned to a journal object, e.g. &JDocJ1. &JDocJ2.)

&Dn – Document Interchange Journal (where **n** is the file alias assigned to a DIJ object, e.g. &Ddij1.)

&C. – pages generated to the point at which the command is called

CommandBefore

optionally identifies a system command that will be executed prior to the start of processing application data with DOC1GEN. See Commandnn for more information.

CommandOK

optionally identifies a system command to be executed only after the application data has been processed successfully by DOC1GEN. See Commandnn for more information.

CommandFail

optionally identifies a system command to be executed only when the application data fails to be processed successfully by DOC1GEN. See Commandnn for more information.

CommandEnd

optionally identifies a system command to be executed only after the application data has been processed successfully by DOC1GEN. See Commandnn for more information.

AbortOnFail

this defines the behavior of DOC1GEN if a print job run in Server Mode fails. The default **True** will abort Server Mode, while **False** gets Server Mode to attempt to discard the failing print job and await further commands.

OPS file format

Example:

This shows the application of user defined parameters in an OPS file with a corresponding DOC1SBMT program:

```
<Server>
CommandQueue=Host:wuk03.6002
Command00=echo Job &0. succeeded. Pages &C.
Command01=echo Job &0. failed

Command02=&0.


doc1sbmt -i data002.dat -p output1=job240.afp -cmdok 00 MyJob -cmdfail
 01 MyJob 02 OnFailure.cmd
```

Running Server Mode

Start-up

Submit DOC1GEN as normal. The CommandQueue keyword in the **Server** section of the OPS file used when starting DOC1GEN uniquely identifies a Server Mode instance.

See **Working with Generate** on page 7 for details of preparing and submitting a DOC1GEN application.

Processing Application Data and Associated Commands

> **Note:** Note that the '&' character is a reserved character and must not be used when specifying paths or names.

Use the DOC1SBMT program to submit application data to an instance of DOC1GEN in Server Mode. Details of this program follow later in this section.

Using DOC1SBMT you must tell server mode:

- the command queue to submit to – i.e. the particular instance of DOC1GEN Server Mode
- the location/name of an application data file to be processed
- the location/name of the printstream file to be created

You may also need to tell Server Mode:

- the locations/names of any Journal files to be generated by the application
- system commands to be generated before the application data is processed; if it processes OK; or if it fails to process
- the locations/names of files to receive overflowed datastream/journal data if using dynamic page numbering (refer to **Working with Generate** on page 7 for more information).

Stopping Server Mode

The DOC1QUIT program is used to stop a particular instance of DOC1GEN Server. Details of this program follow later in this section.

mknod (UNIX only)

Purpose:	Creates a UNIX pipe which can be used with DOC1GEN Server Mode.

Preparation:

Although it is established in advance, the name of the pipe must also be specified in the **Server/CommandQueue** keyword of the OPS file that is used when starting Server Mode.

mknod is executed via the UNIX command line.

Syntax:

```
mknod QueueID p
```

Parameters:

QueueID

is a unique path name of the pipe to be created. IMPORTANT: this name must also be specified in the **Server/CommandQueue** keyword of the OPS file that is used when starting Server Mode.

DOC1SBMT under UNIX or Windows

Purpose:

Submit application data to an instance of DOC1GEN Server Mode.

Preparation:

DOC1SBMT is run from the command line of the appropriate operating system.

Note that each of the keywords for DOC1SBMT is preceded by a minus sign and then the keyword itself (or a valid abbreviation) followed by a space and then parameters (if any). All keywords/parameters should be on one command line, separated by spaces if necessary.

Syntax:

```
doc1sbmt -input Name -print Outputn=Name
{-cmdqueue QueueID | -socket address:port | -host hostname:port }
[ -journal JournalList -dij Name -cmdbefore Command# [ParmList] -cmdok Command# [ParmList]
-cmdfail Command# [ParmList] -cnfsetting SectionList -jobchange hip ops
```

Parameters:

-input

Name identifies the path/file containing the application data to be processed. Keyword can be abbreviated to -i.

-print

Outputn identifies to which file the printstream is being sent. The application data can be output to multiple printstreams. Name identifies the path/file to hold the printstream generated by this submission to Server Mode. Each entry should be separated with a comma or a space. Keyword can be abbreviated to -p.

-cmdqueue	Where submission is to a Server Mode process specified as a named pipe QueueID identifies the appropriate pipe. Keyword can be abbreviated to -queue, -cmd or -q.***
-socket	Where submission is to a Server Mode process specified as a TCP/IP socket (using ‘dot’ addressing) address:port identifies the appropriate socket. Keyword can be abbreviated to -s. ***
-host	Where submission is to a Server Mode process specified as a TCP/IP socket (using ‘host name’ addressing) hostname:port identifies the appropriate socket. Keyword can be abbreviated to -ho.***
-journal	JournalList identifies the journal files required by the application (if any). Each journal is identified as a path/file and should be separated by a comma or a space. A maximum of 8 journal files can be specified. Keyword can be abbreviated to -j.
-dij	Name identifies a file that will receive the Document Interchange Journal (DIJ) where this index type is required by your Generate environment. Each entry should be separated with a comma or a space.
-cmdbefore	Optionally identifies a system command that will be executed prior to processing application data with DOC1GEN. Command# relates to the commands specified in the Server section of the OPS file. ParmList identifies up to 10 parameters to be passed to the command each separated by a space. Keyword can be abbreviated to -before, -cmdb or -b.
-cmdok	Optionally identifies a system command to be executed only after the application data has been processed successfully by DOC1GEN. Keyword can be abbreviated to -ok or -cmdo. See CMDBEFORE above for more information
-cmdfail	Optionally identifies a system command to be executed only when the application data fails to process successfully by DOC1GEN. Keyword can be abbreviated to -fail, -cmdf or -f. See CMDBEFORE above for more information.
-cnfsetting	SectionList identifies additional processing parameters to be associated with the application (if any). Each item in the SectionList corresponds to particular section in the OPS file and is made up of Section, Keyword and Parameter settings. Refer to the Designer User’s Guide for more information about OPS settings. Keyword can be abbreviated to -cnfs
-jobchange	Optionally, instructs DOC1GEN to unload the current HIP/OPS and reload a new HIP/OPS pair. Keyword can be abbreviated to -job
-help	Optional, provides a description of the command format. Keyword can be abbreviated to -he or ?.

***	You must specify only one of -cmdqueue, -socket or -host parameters. This should reference the queue name or TCP/IP socket used as the parameter for the CommandQueue keyword in the OPS file used when starting the appropriate Server Mode process.

Example:

Simple submission to a named pipe under UNIX for an application using no journal files and no command requirements:

```
doc1sbmt -input data001.dat -print
output1=gen/output/job1.out,output2=gen/output/job1.ps,output3=gen/output/job1.pdf
-cmdqueue temp/doc1genq
```

Same submission to a TCP/IP socket (‘dot’ addressed):

```
doc1sbmt -input data001.dat -print outpu1=gen/output/job1.out -socket 10.133.54.202:6000
```

Same submission to a TCP/IP socket (‘host name’ addressed):

```
doc1sbmt -input data001.dat -print output1=gen/output/job1.out -host spa02:6000
```

Using all keywords (with abbreviations) – most files assumed to be in current directory:

```
doc1sbmt -i data002.dat -p output1=job240.afp,output2=job300.afp -q doc1genq_2 -j j002.j1
j002.j2 -dij j003.jrn -b 01 date time -ok 02 -f 03 DOC1GEN SM failed -cnfs <KeyMap>map1=
km1.xml -job j001.hip j001.ops
```

DOC1QUIT under UNIX or Windows

Purpose:	Closes an instance of DOC1GEN Server Mode.
Preparation:	DOC1SBMT is run from the command line of the appropriate operating system.

Syntax:

```
doc1quit {-cmdqueue QueueID | -socket address:port | -host hostname:port
 }
```

Parameters:

-cmdqueue	Where the Server Mode process is specified as a named pipe QueueID identifies the appropriate pipe. Keyword can be abbreviated to -queue, -cmd or -q.

-socket	Where the Server Mode process is specified as a TCP/IP socket (using ‘dot’ addressing) address:port identifies the appropriate socket. Keyword can be abbreviated to -s.
-host	Where the Server Mode process is specified as a TCP/IP socket (using ‘host name’ addressing) hostname:port identifies the appropriate socket. Keyword can be abbreviated to -ho.

Example:

```
doc1quit -q /temp/doc1genq
doc1quit -socket 10.133.54.202:6000
doc1quit -ho spa02:6000
```

4 - Running Generate as a Started Task

In Started Task mode Generate runs continuously on a z/OS system with a particular Generate production environment loaded. It automatically selects and processes jobs when a dataset is passed to its defined input channel and passes the resulting output datastream(s) to the defined output channel. The input and output channels can be either JES output queues, MQSeries pipes or, in the case of output only, disk datasets.

If required, new job environments can be loaded dynamically according to controls in the input data or you can simply run multiple versions of the Started Task with different input channels defined.

The Started Task environment is controlled via a configuration file which identifies the required production job and input channel, and maps the attributes of the input channel with one or more output channels.

In this section

Requirements

These additional modules are supplied for use with Started Task:

- DOC1ST controls start-up
- LYXMSGTB contains run-time messages
- LYXPSO interfaces with the DOC1GEN program and the JES subsystem

You will also need IBM MQSeries installed if you intend to use this as an input or output channel.

The link to DOC1GEN is dynamic and you can therefore replace the DOC1GEN program with a later version without re-linking with Started Task.

All libraries that are defined in STEPLIB for Started Task need to be included in an APF authorized load library so that it can interface with the JES sub-system. Consult your system programmer for more information.

Defining the environment

A single configuration file supplies the settings required to initialize a Started Task. This is a text file that can be created and edited using the standard system editor.

The configuration file must have three sections: Selection Criteria – keywords that define the attributes by which input is selected from the channel and which provide run-time settings; Selection Groups – one or more SELECT groups of keywords that map specific attributes in the input channel with a named output group. Output Groups – one or more OUTPUT groups of keywords that link to an output label and identify the required attributes of an output channel.

A simple example follows. Note the use of parenthesis around the section names and the SELECT and OUTPUT keywords plus unique names to mark the start of specific groups.

```
(Selection Criteria)
 CLASS=QY
(Selection Groups)
 SELECT AFP
  CLASS=
  OutGroup=TO.AFP
 SELECT AFPandPDF
  CLASS=Y
  OutGroup=(TO.AFP,TO.PDF)
 SELECT Default
  OutGroup=TO.PDF
(Output Groups)
 OUTPUT TO.AFP
  DDNAME=AFPOUT
  CLASS=A
 OUTPUT TO.PDF
  DDNAME=PDFOUT
  CLASS=B
```

The order of selection groups is significant – starting from the top the first group that matches the criteria of the incoming dataset will be used. A ‘Default’ selection group should normally be coded at the bottom of the section to cater for selected datasets that do not match the criteria of other groups.

The OutGroup keyword within a selection group points to the name of one or more output groups which determine how many types of output datastream are required. If multiple output groups are referenced you must code them in parentheses.

Output groups themselves must contain a DDNAME keyword the value of which matches an output label specified either at job creation time or within an OPS file used when initiating the Started Task. This link determines the type of output datastream(s) to be generated.

Depending on the output datastream being generated you may also want to add keywords that instruct specific DCB attributes for output datasets. RECFM, LRECL and BLKSIZE are supported for this purpose as in the following configuration file fragment:

```
...
(Output Groups)
 OUTPUT TO.AFP
  DDNAME=AFPOUT
  RECFM=VBA
  LRECL=8025
  BLKSIZE=8025
```

Using JES output queues

When using JES queues as an input channel the selection criteria specified in the configuration file is used to identify the high level attributes of SYSOUT datasets intended for Started Task. The selection groups can then be used to select datasets according to more detailed JES criteria if required and direct the results to the appropriate output groups.

> **Note:** Datasets that are 'held' will never be selected by Started Task regardless of whether they match the selection criteria or not.

Output groups themselves define the attributes of the new SYSOUT dataset that will contain the job output. Note that output datasets automatically adopt the job name and number of the Started Task process when they are placed on JES queues.

Within the configuration file sections these criteria are mostly specified exactly as they would appear as JCL attributes; for instance CLASS=A or FORM=G129A. The exceptions to this are:

- a CLASS attribute in Selection Criteria may include 1-8 classes to allow a broader initial selection. For example CLASS=ABF1.
- most criteria in Selection Groups can include a wildcard asterisk indicating that the remaining part of the attribute name is variable. For example DEST=ROOM1* or FORM=F1*.
- you can specify an asterisk for any attribute in an Output Groups indicating that the relevant setting is to be the same as in the input dataset.

Summary of supported SYSOUT attributes

Selection Criteria	`CLASS, EXTWTR, DEST` Where the SYSOUT API is installed you may also use: `JOBNAME, FORM, PRMODE, FCB, UCS, FLASH`
Selection Groups	`CLASS, EXTWTR, DEST, JOBNAME, FORM, ROOM, CREATOR`
Output Groups	`CLASS, FORM, EXTWTR, DEST, FCB` You may also reference an `OUTPUT` card specified in the JCL used to initiate Started Task. Output attributes are then assumed from this card.

The following example of a configuration file summarizes these points:

```
(Selection Criteria)
 CLASS=YZ
 DEST=ROOM23
 PRMODE=PAGE
```

```
(Selection Groups)
 SELECT XClass
  CLASS=X
  FORM=INV*
  OutGroup=OUT.ACLASS
 SELECT YClass
  CLASS=Y
  JOBNAME=G1546*
  OutGroup=OUT.BCLASS
 SELECT Default
  JOBNAME=’*’
  OutGroup=OUT.REFER

(Output Groups)
 OUTPUT OUT.ACLASS
  DDNAME=OUTPUT1
  CLASS=A
  FORM=D21A
 OUTPUT OUT.BCLASS
  DDNAME=OUTPUT2
  CLASS=B
  DEST=’*’
  EXTWTR=’*’
 OUTPUT OUT.REFER
  DDNAME=OUTPUT3
  OUTPUTCARD=OUTREF
```

Using MQSeries

When using MQSeries as an input channel you will actually need to work with two types of concurrent MQSeries queues:

- The actual data queue(s) to which input data files are added.
- A ‘trigger’ queue which notifies Started Task that input data is ready to be processed and in which queue it can be found.

The trigger queue must exist before initiating Started Task and it must stay active at all times. The trigger queue is identified to Started Task using the TriggerQ keyword in the main selection criteria of the configuration file as in the following fragment:

```
(Selection Criteria)
 TriggerQ=MQ.INPUT

(Selection Groups)
 SELECT FromMQ
  OutGroup=TO.MQ
...
```

To notify Started Task that input data is ready to be processed a record must be added to the trigger queue using the following format:

```
GEN SELECT=GroupName,QNAME=QueueName
```

Where: GroupName indicates a selection group in the Started Task configuration file that will handle the data; QueueName indicates the MQSeries queue where the input data is waiting.

So, for example, your trigger record might contain:

```
GEN SELECT=FromMq,QNAME=DOC1INPT
```

You may also use an MQSeries queue to receive the resultant output datastream(s). To do this the relevant OUTPUT group must contain the keyword MQNAME=QueueName as in the following fragment.

```
...
(Selection Groups)
 SELECT MQOUT
  OutGroup=TO.MQ
(Output Groups)
 OUTPUT TO.MQ
  MQNAME=GEN.TO.MQ
...
```

Both input and output queues must either pre-exist or be created by the client program submitting data to Generate. Started Task will never destroy any of the Queues involved although the input and output queues may safely be destroyed by the client program if required.

Started Task uses the notion of ‘staging files’ to temporarily hold the datasets coming from or going to MQSeries queues. Keywords in the selection criteria section of the configuration file allow you to specify the high level qualifier for the names of the datasets to be used for this purpose. Other keywords allow you specify the attributes of such datasets. If these settings are not defined Started Task will attempt to use SYSDA workspace with a default dataset HLQ of ‘DOC1’. Dataset attributes will also assume default settings. The full list of keywords related to staging files are:

```
* Settings for input staging
MQIHLQ=dsn
MQIUNITS=CYL|TRK
MQIPRIMARY=nnn
MQISECONDARY=nnn
MQIUNIT=device_or_group
MQIVOL=volser
MQIBLKSIZE=nnnnn
MQIRECFM=F|V|U,B,S,T,A|M
(* Settings for output staging
MQOHLQ=dsn
MQOUNITS=CYL|TRK
MQOPRIMARY=nnn
MQOSECONDARY=nnn
MQOUNIT=device_or_group
MQOVOL=volser
MQOBLKSIZE=nnnnn
MQORECFM=F|V|U,B,S,T,A|M
```

When the relevant process is successfully completed the staging datasets are deleted. For input files, if Started Task fails to process the dataset it is left on the staging location to allow the user to take remedial action if required. You may need to manually delete such files from time to time.

The following is an example of a configuration file where Started Task is using MQSeries as both input and output channels:

```
(Selection Criteria)
 TRIGGERQ=MQ.INPUT
 MQIHLQ='G1097.GEN.INSTAGE’
 MQIUNITS=TRK
 MQIPRIMARY=50
 MQISECONDARY=100
 MQIUNIT=DISK
 MQIVOL=PR9801
 MQIBLKSIZE=8205
 MQIRECFM=VB
 MQOHLQ='G1097.GEN.OUTSTAGE’
(Selection Groups)
 SELECT SYSandMQ
  OutGroup=(TO.SYSOUT,TO.MQ)
 SELECT SYSOUT
  OutGroup=TO.SYSOUT
(Output Groups)
 OUTPUT TO.SYSOUT
  CLASS=A
  FORM=D21A
 OUTPUT TO.MQ
  MQNAME=GEN.TO.MQ
```

Using datasets as output channels

It is possible to write the output files created by Started Task jobs directly to new or existing datasets if required.

To configure such output the Dataset keyword of an output group is used to identify the recipient dataset. If a fixed dataset name is supplied Started Task will attempt to append all output to the same file.

More usually however, you will want to use variable dataset names to add individual outputs to separate datasets or members. To do this you will need to supply part of the dataset name string within the first record of input data delivered to Started Task for each job. In this scenario the Dataset keyword acts as a template for the eventual dataset name and identifies where the variable part of the name can be located within the input data. it has the following format:

```
Dataset=base_name{offset,length}
```

Where offset and length identify the location of the bytes within the first record that contain the dataset name variable string. For instance, in the following example dataset names in the format G1ACC.GEN.variable will be used and the variable string is provided in bytes 6-13 of the first record of each input file:

```
Dataset=G1ACC.GEN.{6,8}
```

If you want to create new datasets for each output you will normally also need to use a range of other supported keywords in the output group that allow you to define their attributes. The following configuration file fragment contains examples of all the supported attributes:

```
...
(Output Groups)
 OUTPUT AFP.TO.DATASE
  Dataset=MX2511.AFPDS.{15,8}
  Status=NEW
  Disp=KEEP
  Units=CYLS
  Primary=1
  Secondary=5
  Blksize=8204
  Volume=MXC001
...
```

Custom selection criteria

If you need to use very specific criteria for identifying different types of input data you can have Started Task interrogate a value in the first record of input datasets. To use this feature two additional keywords must be coded in the relevant selection groups in the following format:

```
DataPos=offset
Data=['string'|X'Hex']
```

Where string or hex is the value that Started Task will look for and offset is a start position in the first input record where the value is to be found. You may code the value either as a text string or as a series of hexadecimal characters in the format shown in the example below.

The DataPos and Data keywords must always be coded as a pair within a selection group. DataPos may indicate a different offset in different groups.

The following configuration file fragment demonstrates how this feature might be used:

```
...

(Selection Groups)
 SELECT Data1
  DataPos=12
  Data=’Type1’
  OutGroup=OUT.ACLASS
 SELECT Data2
  DataPos=12
  Data=’Type2’
  OutGroup=OUT.BCLASS
 SELECT Data3
  DataPos=25
  Data=X’D499A240C29996A695’
  OutGroup=OUT.CCLASS
...
```

Switching production jobs

Normally a particular Started Task will process a single Generate production environment. In this case the job is specified according to the HIP file indicated in the JCL used to initiate Started Task.

If required however, you can have a single Started Task switch between production job environments according to the selection group that is activated.

To use this feature you will need to create Override Production Settings (OPS) files for each job environment that can be loaded. The OPS file must identify the input/output file to be used with the job. The format of the OPS file is described in the main Designer User's Guide.

Note: It is the users responsibility to ensure input data is suitable for the currently loaded job.

Once you have created your OPS files you must reference each of them and the required HIP files in DD statements in the JCL used to initiate Started Task. These DD statements can then be referenced using PARM keywords –which can have a maximum of 8 characters – within the appropriate selection groups. When such a group is matched according to its selection criteria the relevant production job environment will be loaded if it is not already active. It will then stay loaded until a selection group with a different PARM is invoked.

So, for example, your JCL may contain:

```
//DOC1MHIP DD DSN=GEN.HIPS(HIP1)
//DOC1OPS1 DD DSN=GEN.RUN(OPS1)
//DOC1OPS2 DD DSN=GEN.RUN(OPS2)
```

Your selection groups might be coded as follows:

```
...

(Selection Groups)
 SELECT Normal.Job
  CLASS=X
  OutGroup=OUT.ACLASS
 SELECT Alternate.Job
  CLASS=Y
    PARM=DD:DOC1MHIP,OPS=DD:DOC1OPS1
  OutGroup=OUT.BCLASS
 SELECT Default
    PARM=DD:DOC1MHIP,OPS=DD:DOC1OPS2
  OutGroup=OUT.REFER
...
```

Dealing with failures and specifying subsequent processing

You can use the Continue keyword in the main selection criteria of the configuration file to specify how Started Task handles input datasets that do not process successfully.

- Continue=Yes The dataset that caused the failure is suspended; i.e. if SYSOUT it is held, if emanating from MQSeries the staging file is not deleted. A system message is issued on the system console and Started Task will continue processing. This is the default setting.
- Continue=No Started Task processing will be suspended (but the task will not be terminated) and an appropriate system message issued.
- Continue=Stop Both the dataset that caused the failure and the Started Task process is suspended.

You can also use the configuration file to specify actions to be taken when a batch of data has been processed by Started Task. A range of keywords allows Started Task to react to either successful or unsuccessful completion of each batch by issuing a message to the system log, issuing a console command and/or submitting a job via a JCL deck in a dataset. You may use none, any or all actions for both conditions as required.

For example, on a failure you could issue a warning message about the condition, issue a console command to hold the queue that Started Task normally uses for input and submit a JCL deck to run programs that process corrective actions.

These actions are coded as optional keywords within a selection group. The available keywords and parameters are:

OnFailureMessage=*'Text and substitution codes'*

OnFailureCommand=*'Text and substitution codes'*

OnFailureSubmit=*'DSN referencing a JCL deck'*

OnCompletionMessage=*'Text and substitution codes'*

OnCompletionCommand=*'Text and substitution codes'*

OnCompletionSubmit=*'DSN referencing a JCL deck'*

You may include any of the following substitution codes where appropriate to the action.

&jobname	Name of job providing processed data
&jobid	Job number of job providing processed data
&date	System date (yyyymmdd)
&time	System time (hhmmss)

&hour	Hour from system time (24 hours)
&min	Minute from system time
&sec	Second from system time

Note: Date and time values are set when a batch is initially selected from the input channel and remains static.

In the following example an information message is issued when a batch is successfully processed. If a failure occurs processing is suspended (Continue=No) and a clean-up job is submitted using the OnFailureSubmit command:

```
(Selection Criteria)
 CLASS=Q
 Continue=No
(Selection Groups)
 SELECT All.Jobs
  CLASS=X
  OutGroup=OUT.ACLASS
  OnCompletionMessage=
  ’COMPLETED- &jobname &jobid &date’
  OnFailureSubmit=’G132.GEN.FJCL’
...
```

Running Started Task

The Started Task environment is controlled via a configuration file which identifies the required production job and input channel, and maps the attributes of the input channel with one or more output channels.

JCL and start-up

Start-up JCL for a Started Task application is identical to a regular DOC1GEN application with the following exceptions:

- you do not need to specify an input file
- the Started Task configuration file must be identified using JCL label DOC1CONF
- you may need to reference multiple HIP files if you intend to switch dynamically between production jobs.
- if you intend to use MQSeries as input or output channels you may need to reference the MQSeries load libraries SCSQLOAD and SCSQAUTH in your STEPLIB concatenation.
- segmented resource HIP files are not supported in Started Task.

Refer to **Working with Generate** on page 7 for full details of the files that make up a complete DOC1GEN run-time environment.

If you intend to run multiple, concurrent versions of Started Task the following rules apply:

- the start-up procedures will need to be stored under different member names.
- each version must use a different configuration file.
- the selection criteria specified in any two configuration files must not be the same.

Process

Once a Started Task is initialized it will remain active until it is stopped by an operator command (see below) or cancelled.

It will poll its input channel using the default frequency or using the frequency specified for the Interval keyword in the selection criteria of the configuration file. For example:

```
(Selection Criteria)
```

```
 Interval=500
...
```

Interval is specified in terms of thousand’s of a second and is a value between 100-1800.

Input datasets are selected for processing automatically according to the selection criteria in the configuration file. If there are multiple datasets matching the selection criteria they will be selected one at a time.

If the generation process completes successfully the output dataset is created with attributes according to the output group that was used. The original dataset is deleted either from the JES queues or from the staging files if you are using MQSeries.

Operator Commands

Provided Started Task is being run from an authorized procedure library, four operator commands allow you to control and modify the Started Task environment.

S procname	Starts a Started Task environment whose start-up JCL is stored in procedure procname.
P procname	Stops the Started Task environment started by the above command.
F procname, STOP	Temporarily suspends the selection of datasets for processing by Started Task.
F procname, START	Restarts input dataset selection for a Started Task that was previously suspended manually with the above command or automatically following a problem with the DOC1GEN run-time environment when processing an input dataset.

You may want to suspend processing if you are experiencing problems with the DOC1GEN run-time environment. The "F procname, START" command re-initializes the DOC1GEN run-time environment and any changes to this environment made since the "F procname, STOP" command was issued will be affected by the START command. Note that this affects only the DOC1GEN environment and that you cannot change the Started Task environment itself (i.e. that specified in the configuration file) without stopping and restarting the procedure.

Extended configuration file examples

```
*************************************************************************
              Started Task configuration file for JES queues
************************************************************************
(Selection Criteria)
 Interval=500 Polling interval is .5 of a second
 Continue=No  Suspend processing on error
 EXTWTR=GEN* Select if writer name is prefixed’GEN’ and...
 CLASS=ABY is A, B or Y class SYSOUT

(Selection Groups)
 * If A class with jobname prefix of "Test" two types of output are required
 SELECT A.Class
  JOBNAME=TEST*
  CLASS=A
  OutGroup=SYSOUT.HCLASS,DISK

 * All Y class invokes an alternative DOC1GEN environment
 SELECT Y.Class
  CLASS=Y
  PARM=DD:HIP2,OPS=DD:JOB2OPS
  OutGroup=SYSOUT.HCLASS

 * Select if first record of input dataset contains ‘TYPE1’ at offset 10
 SELECT By.Data
  DATAPOS=10
  DATA='TYPE1'
  OutGroup=DISK

 * Default group issues messages if invoked
 SELECT Default
  JOBNAME=*
  OnCompletionMessage='Default direction successful - &Date &Time'
  OnFailureMessage='Default direction FAILED - &Jobname &JobID'
  OutGroup=SYSOUT.OTHER
(Output Groups)
 * Send AFP output to H class & Dest R901
 OUTPUT SYSOUT.HCLASS
  CLASS=H
  DEST=R901
  DDNAME=AFPOUT
  RECFM=VBA
  LRECL=8025
  BLKSIZE=8025

 * Store PDF output on disk - dsn variable provided at offset 22 in 1st record
 OUTPUT DISK
  DDNAME=PDFOUT
  DATASET='G1UK.GEN.AUTO.{22,8}'
  DISP=KEEP
  STATUS=SHR
  UNITS=BLK
  PRIMARY=300
  SECONDARY=300
  BLKSIZE=8025
  VOLUME=USER95

 * Copy DEST & EXTRWTR from input datasets, all other attributes from an OUTPUT
 card
 OUTPUT SYSOUT.OTHER
  DEST=*
  EXTWTR=*
  OUTPUT=TESTOUT

*************************************************************************
            Started Task configuration file for MQSeries pipes
************************************************************************
(Selection Criteria)
 TRIGGERQ=MQ.INPUT
 * Set workspace for input staging files
 MQIHLQ='G1097.GEN.INSTAGE'
 MQIUNITS=TRK
 MQIPRIMARY=50
 MQISECONDARY=100
 MQIUNIT=DISK
 MQIVOL=PR9801
 MQIBLKSIZE=256
 * Set workspace for output staging files
 MQOHLQ='G1097.GEN.OUTSTAGE'
 MQOUNITS=CYL
 MQOPRIMARY=5
 MQOSECONDARY=10
 MQOUNIT=DISK
 MQOVOL=PR9801
 MQOBLKSIZE=8205

(Selection Groups)
 SELECT SYSandMQ
  OutGroup=TO.SYSOUT,AFP.TO.MQ
 SELECT SYSOUT
  OutGroup=TO.SYSOUT

(Output Groups)
 OUTPUT TO.SYSOUT
  CLASS=A
  FORM=D21A
 OUTPUT AFP.TO.MQ
  MQNAME=GEN.TO.MQ
  RECFM=VBA
  LRECL=8025
  BLKSIZE=8025
```

5 - Programming PCE

The Post Composition Engine (PCE) handles the requirements of some applications for additional manipulation of output datastreams once they have been created by Generate. PCE is typically used for such things as merging datastreams, reordering pages and adding new presentation objects to existing composed pages.

In this section

The PCE environment

PCE supports the manipulation of AFP and PostScript datastreams as produced by Generate. An individual PCE job should only attempt to manipulate one type of output datastream and all datastreams must have been composed using the same output device settings (as specified when publishing a job in the Designer). PCE can handle any of the file structures produced by Generate including the VSAM formats that can be used under z/OS.

PCE's basic unit of work is a composed page within an output datastream. You can process a datastream on a page-by-page basis or you can use journal file created by Generate to act as an index into specific pages that need to be manipulated.

> **Note:** See the Designer User’s Guide for information about setting up your application to create journals.

A PCE job is programmed using a script file. A script is coded using a simple proprietary language that can be produced in any standard text editor.

PCE also requires an Initialization File (INI) to be created and specified when starting the program. This allows you to identify the script to be processed, the type of datastream being manipulated, plus environmental information about the system on which the job is to run and any customization for the output datastream if new elements or pages are being created. You can also add symbols to the INI that can be referenced within the script.

The PCE program itself is called DOC1PCE on all platforms. You will need to execute this program from the batch environment appropriate to your production system: command line, script, JCL etc.

Once you have created your PCE resources refer to **Running PCE** on page 200 for details of actually running the job.

Function overview and script command summary

The PCE script language consists of a set of commands most requiring one or more parameters. Each command starts on a new line and must be terminated by a semi-colon. Parameters can be supplied as literal expressions or from the contents of a variable.

The highlighted words in this section indicate actual PCE commands. Details of each command can be found in the **Composition Edit Commands** on page 172 .

Procedures and program control

To avoid repetition script commands can be grouped into procedures. All scripts must have at least one procedure known as 'Main'. This will be assumed to be the first procedure in a script if it is not explicitly defined. You must use the **declare procedure** command to identify all procedures (including Main) before using any procedural statements.

The start of a procedure is identified by a **begin procedure** on page 87 command. All statements from this command up till the next **end procedure** on page 106 command are considered to be part of the procedure.

A procedure can invoke a lower level procedure via the **call procedure** on page 88 command. Procedures cannot be called recursively, i.e. they cannot call themselves or a higher level procedure.

If required, the **return** on page 152 command in a procedure can immediately pass control back to the statement following the appropriate **call procedure** on page 88 command. Note that **end procedure** on page 106 implicitly performs the same function.

Code that is used more than once, say in different applications, can be written as separate scripts and then used in a main script by using an **include** on page 117 command. Include files can be referenced anywhere in a script. At run-time the include command is replaced by the contents of the specified file.

Loops

Standard program repeat features are available via **begin loop** on page 86/**end loop** on page 105**exit loop** on page 108 and **for…next** on page 112 constructs.

Conditions and branching

Number values can be compared using standard operators (eq, gt, le, etc.) and text strings using the **equals** on page 107 or **contains** on page 92 functions. The results of such comparisons produce a TRUE/FALSE result which should be stored in a variable.

Program branching can then be achieved by using such results of part of an **if…else…end if** on page 116 construct.

Variables

A variable can be used to store any type of data that is accessible to a PCE script including strings, numbers and actual pages of composed output datastreams.

You can define arrays of variables to store multiple examples of the same data type. You may, for instance, want to store multiple pages of output datastream under the same name but be able to manipulate them independently.

All variables must be **declared**d before they are referenced by other commands within a script. The declare command assigns a name by which the variable is referenced in the script. All variables have global scope and you can reference any variable within any procedure providing it has already been declared.

Values are assigned to variables by **read** on page 145ing data into them from a file or by using the **Let** command. You can use let to assign literal values, store the result of functions and copy values from other variables.

Typing of variables is automatic and is assumed from the assigned data. When reading from file into a variable the data is assumed to be text unless a complete page of output datastream is being read. Using a function to assign a value explicitly sets the target variable to the appropriate data type.

There is no fixed size limit to a variable and the amount of memory required to store variables is self-defined by the data assigned to them. However, PCE does not perform memory checking and you should therefore be aware of the amount of data your script requires to be held in memory at any one time and ensure that your system has sufficient resources to cope. You can release memory that has been allocated by a variable using the **release** on page 150 command. This is particularly useful when variables are used to many store pages of output datastream.

Up to 1900 variables can be used by a single PCE script. When specifying array variables however, bear in mind that each array element accounts for an individual variable allocation.

Symbols

Symbols that have been defined in the PCE initialization file or on the start-up command line can be accessed by the script file by using the **symbol** on page 158 command.

File handling

Files used by a typical PCE script consist of the existing output datastream files that are to be processed, any journal files that provide indexes for the datastreams and output files to receive the reorganized datastreams or other information.

The **open** on page 136 command is used to identify a file to be used by a PCE script. This assigns a unique ID which is used to reference the file elsewhere in the script.

> **Note:** PCE can open files for read or write but not both simultaneously. A PCE script therefore typically involves reading from one output datastream file, manipulating in memory as required and then outputting to a new file of the same type.

The **open** on page 136 command also requires two type parameters that are crucial to correct processing of a file by PCE. The first parameter defines the record construct of the file and the second specifies its content, i.e. what type of output datastream or other data it contains. Together these parameters dictate how much data should be handled each time a **read** on page 145 or **write** on page 169 command is performed. For instance, an AFP file on an z/OS system should typically be opened as type RECORD/AFPDS and a text journal under Windows will typically use LINE/PLAIN.

If the record construct of your file is not catered for by one of the predefined keywords you can use the format parameter to indicate a custom construct.

> **Note:** If the format parameter indicates additional block or record header data other than that normally used for an output datastream this will be stripped from the file before the pages are processed. When the pages are written to a new file the additional header structures will only be applied if they are explicitly defined as part of the format parameter in the appropriate open command, or if the default format for the file type automatically applies such headers.

Reading and writing

As indicated above, the amount of data handled by the **read** on page 145 or **write** on page 169 command depends on how it has been opened. For instance, an read/write item from a file that has been opened as an output datastream will consists of all the data that makes up a single composed page.

A **read** on page 145 command specifies the number of items to be read and a variable to contain the results of the function. Each time the read is performed all data required to make up the required number of items is stored in that variable. If more than one item is specified and the variable has been declared as an array, each item will be stored in an individual array element. If not, multiple items will be concatenated together.

Similarly the **write** on page 169 command copies items from a variable to an output file. Note that multiple items can only be written by a single write command if the variable has been specified as an array.

File pointers and offsets

Every file has a pointer that indicates the place in the data where the next read or write operation will commence. Every time a read or write is performed the appropriate file pointer is updated.

When you are reading from file you can indicate a specific file offset at which the read should start. You may have stored such offset information in journal file created by Generate.

When you are writing output datastream output you can use the **pageoffset** on page 143 command to access the current file pointer value. This is particularly useful if you are updating pages of output datastream and need to be aware of new offsets as the pages are written back to file.

Resource handling and file merging

Depending on the options chosen for a particular production job the font and image resources required to actually print/present publication output may be stored:

- within the HIP file used to control a publication in the production environment
- within the file header of the actual output datastreams produced by Generate
- in both or neither of the above.

Output datastreams created by the same PCE job will always reference a single set of resources. If you are working with multiple datastreams (and in particular if you intend to merge datastreams into a single file) it is important to define where PCE should look for resources to be used in new output files. The get resource command is provided for this purpose.

If this is not coded PCE will use the resources stored in the first output datastream file to be read by a script. For this reason it is imperative the get resource is coded prior to any file read commands if you intend to use it.

Many output datastream formats reference fonts according to the order they appear in the file header. When you are merging pages from multiple sources this order will often be different so PCE needs to cross-reference the font references to keep a consistent order in the merged output file. If one of the input files holds all the required font references it can act as the master font list and pages from this file can be written to the output file without manipulation. When working with pages from other files PCE must create a cross-reference to the master (known as a font map), look-up the relevant font references for each page and then rewrite the page using the new references. To reduce the impact of this you can specifically define the master input file using an extension of the get resource command.

Using font references

When you intend to add new text to existing pages (see **"Changing composed pages" on page 54**) you may want to use the font command to gather details of a specific font before attempting to reference it. When doing so you can use the number of fonts command as part of a routine to iterate over each available font.

Postscript Open Type Font (.OTF) font handling

When reading or writing PostScript containing PostScript Open Type fonts, the `GET RESOURCES FROM HIP` command must be coded in the PCE script before any `OPEN` command to allow the correct font references to be generated.

> **Note:** PCE does not support the addition of text using CE `=STL` for PostScript Open Type fonts.

AFP Outline Type1 fonts

When using PCE to add text using AFP Outline Type1 font references, the `GET RESOURCES FROM HIP` command must be coded in the PCE script before any `OPEN` command to allow the correct font references to be generated.

Journal files

When your application has a known post-processing requirement it is usual to specify one or more journal files as part of the application design. Journals are typically used to provide an index for the documents and pages within composed datastreams and can be used to allow you to locate particular parts of the output that need further processing. If you include a vector environment variable within the journal you can use this to move the file pointer associated with a datastream to a particular document/page without the need to read all intervening data.

> **Note:** Indexed access to datastreams stored in a VSAM formats always require the presence of vector file offsets in the appropriate journal files so that the required pages can be located.

If you make changes to a datastream (for instance, if new presentation elements are added to pages) you will also normally want to have PCE create a new journal to reflect the amended page offsets.

Vault & DIJ files

If your output datastream is intended to be stored in a Vault Server the application should always generate a specialized index known as a Document Interchange Journal (DIJ). A DIJ is an XML journal which has a fixed structure with a predefined set of elements pertaining to the documents within the datastream. A DIJ always contains one entry for each document in the datastream with which it is associated.

Such files must be **open**ed as type DIJ so that PCE knows how to handle their XML structure. You must use the **read…DIJentry** and **write DIJentry** command when performing IO with DIJ files. You may only read and write a single DIJ record at a time.

The **DIJelement** command allows you to read the individual values stored within a DIJ record. If required, you can use the **change DIJelement** command to update these values.

Documents that are referenced by a DIJ always have a unique identifier stored within the first page which is used for integrity checking when the relevant datastream is loaded into the Document Repository. If your PCE script has cause to remove or reorder the position of pages within a document you may need to use the **add document id** command to ensure that page 1 of each document has a valid identifier. The **document id** command enables you to read the identifier from a page if required.

Data manipulation

A range of features and functions are provided to allow you to manipulate values.

Numbers

Calculations are specified using a standard assignment statement with arithmetic operators: For example: `let <var1> = 6 * 10;`

If you have a number value that has been specifically stored as a string you can convert it to a number type using the **value** function.

Text

The following commands allow you to manipulate and format text strings.

atrim trims leading and trailing spaces

ltrim trims leading spaces

mixc performs mixed case conversion

mapp replaces or inserts text with substitution values from a lookup table.

rtrim trims trailing spaces

string specifically converts a value held as number type to a string type and applies text formatting if required.

substring extracts a number of bytes from a string

symbol returns the value of a symbol from the PCE initialization file or start-up command line

translate uses a specified table in the Generate Translation Tables file to convert a text string to a different code page.

DBCS text

You an specify a double byte character set string by using a the hex encoding format when assigning a value to a variable. For example: `let <var1> = X‘F6,E8,F9,40’;`

Environment data

The following commands allow you to access system values and regional preferences as specified in the PCE initialization file:

date the system date

day the name of a specified day of the week from PCE preferences

monthname a specified month name from PCE preferences

monthabbrev a specified month abbreviation from PCE preferences

numericconvcode the regional settings for standard punctuation characters from PCE preferences

numericpadding the regional character to be used for padding numbers from PCE preferences

ordinal the character to be used for a specified ordinal from PCE preferences

time the system time

uservalue a custom value from PCE preferences.

You can change which <Preferences> section is currently active by using the **set preferences** command.

Changing composed pages

Although PCE handles pages of output datastream as a whole, some restricted editing of the presentation items they contain is possible.

Text strings within composed pages can be amended directly using the **overwrite** or **replace** commands.

The **merge** command can be used to generate a single composite page from the presentation items from two composed pages. The **move** command will offset all printable items on a composed page by the defined values.

Adding new presentation items

The **begin ce**/**end ce** command construct is used to add entirely new items to existing pages. Commands within this construct are known as composition edit (CE) commands and, for greatest efficiency when manipulating the actual output datastream protocol, are defined at a lower level than regular PCE commands. As a result, CE commands have a different general format from regular PCE commands and require greater care in ensuring the syntax and sequence of such commands is correct. The format of CE commands is fully discussed in **Composition Edit Commands** on page 172.

Barcodes

Within a begin ce/end ce construct the Do Barcode command translates a text string into one of a number of supported barcode types with user defined formatting options.

Note: that Font scaling and line drawing barcodes are not supported for PCE.

Printer controls

The **add medium map** command can be used to include an AFPDS Invoke Medium Map structured field to an existing 'page' of AFP output datastream.

For PostScript output files you can use the **set page name** command to add a page name to the output being generated.

Error Handling and Environment Information

Syntax errors in the script and inconsistencies between the script file specifications and actual data will result in an immediate termination of the PCE process.

For errors relating to IO functions you can control the actions to be taken by evaluating error codes. The codes are generated by errors encountered during the processing of the **open**, **close**, **read**, write, replace and **overwrite** commands. The **on error call** function allows you to specify a procedure in the current script File that will be called whenever such a problem is encountered. To be effective this procedure should query the nature of the error and have logic to deal with each anticipated error. If **on error call** is not coded, PCE will terminate and issue a generic error message on encountering an IO error.

You can also specify a customized value to be assigned to the return code issued on completion of a PCE process via the predefined variable name <sys_exit_value>. The value can be assigned via the let (number) command and must be a number in the range 0-999.

Document Groups

Some output datastreams support the notion of individual documents within the stream – two or more pages that are associated in a group structure. When working such streams you may want to place pages within groups or manipulate existing groups.

In the current release document groups can be created for AFPDS only.

> **Note:** You must work with document groups if you you intend to create group level indexes for AFP datastreams.

Normally when you read pages from a datastream any group related fields will be ignored. Pages that need to be manipulated as a document must be specifically added to a document group while in PCE memory. You can do this as directly or indirectly as follows:

Directly from existing pages

Use the **add document name** command specifying a variable that contains one or more pages. A new group will be created containing these pages referenced by the name specified. Note that the pages must not have formed part of an existing group otherwise results will be unpredictable – use the document group method where this is the case (see below). Use the **write** command to append the grouped pages to an output file as required.

If you do not want to read all the required pages into the group at the same time – perhaps because of memory constraints – then you can perform partial reads and writes to build the new group. When using this method you will need to code the **read…document** command specifying start, middle, end or all keywords to let PCE know how to structure the pages within the group. You must also code the **add document name** command twice using the start and end keywords at the appropriate times before writing to the output file.

Indirectly by reading pages into a document group

Use the **read…document** command to copy pages into a document group. If the pages read already form a complete group then this structure will be retained. Use the **add document name** command to give the document group an identifier or overwrite an existing name.

Note: Generate does NOT produce pages within document groups when it creates an output datastream so reading the pages of a ‘document’ produced by Generate will not give you a group structure. Use the write document command to write the grouped pages to an output file.

Support for AFPDS Indexing

Several commands allow the reading, writing and manipulating of AFPDS Tag Logical Element (TLE) structured fields. A TLE has a name by which it can be identified and an index value.

TLEs can be stored either within individual pages or within an AFP document header. When working with document level TLEs you should normally read the pages of a document into a PCE document group structure to ensure the header is included. Any number of TLEs can be present in pages or document headers.

Page level TLEs

To read the value of an existing page level TLEs use the **TLE** command. Use the **TLE add**, **TLE delete** or **TLE replace** commands with the page keyword to adjust the index settings for a page.

Document level TLEs

Before you can use any command related to document level TLEs you must first read the appropriate pages into a PCE document group (see **"Document Groups" on page 55**).

To read existing TLEs in the group use the **document TLE** command. Use the **TLE add**, **TLE delete** or **TLE replace** commands with the document keyword to adjust the index settings within a document header.

If you are using partial read/writes to create a new document group (i.e. you are specifying the start, middle and end keywords as part of an **add document name** group) then you must ensure that the TLE commands are used before the start portion is written to the output file. See the example below for an example of the required sequence of commands.

If required you can also extract from or add pages to an existing document group by using the **extract document page** and **move page** commands. You may want to do this if you need to work with page level TLEs in the original AFP pages that are to make up the group.

Example of sequencing TLE index commands for a document group

```
read 10 pages of document start from file <InFile> into <Group>;
add document name of "Big Group" to <Group> at start;
add document TLE of attrib "first_tle" value "Index1" to <Group>;
write 1 item to <OutFile> from <Group>;

read 10 pages of document middle from file <InFile> into <Group>;
write 1 item to <OutFile> from <Group>;
read 10 pages of document end from file <InFile> into <Group>;
add document name of "Big Group" to <Group> at end;
write 1 item to <OutFile> from <Group>;
```

User Exits

User exits functions are initiated via the **call userexit** command.

User exits of type INSTRUCTION call an external user-defined program function and, optionally, provide a value that can provide input to the PCE script.

User exits of type PRINTSTREAM are intended to return a self-contained segment of output datastream. You can use the **insert object** command to merge such segments into existing pages.

Note: user exits of type DATA_INPUT are not supported for PCE.

Script syntax

All statements are independent and compound statements are not allowed. Thus statements cannot include or be concatenated with other statements.

Every statement must be terminated by a semicolon.

If necessary statements may span multiple lines.

Commands may be written in UPPER or lower case or a MIXture of the two.

Leading and trailing blanks are always ignored including those used in multiple line statements.

Be aware that the script file is always expected to conform to a US English code page (ANSI for Windows and Unix, IBM500 for EBCDIC-based hosts). If your operating system is based on a different code page you may need to make allowances for control characters such as '@'.

Variables and Arrays

All variables must be pre-declared. Variables are identified by token names enclosed in angle brackets. For example:

```
declare <count>;
let <count> = 1;
```

All standard keyboard characters are valid for use as part of the token name except the space character. Names can be any reasonable length but note that the total length of any statement including all token names cannot exceed 999 characters.

The data type and required storage for all variables is defined implicitly by the contents assigned to them and/or the statement used for the assignment. No explicit definition is required. If a variable has been assigned the result of string function but the value is actually a number you can use the **value** command to specifically convert it number type before using it with number functions.

To indicate that a variable is an array specify the number of elements enclosed in parenthesis following the variable name. Reference a specific array element using the same syntax. Element numbers can be specified using a variable if required.

For example:

```
declare <items>(10);
declare <count>;
let count = 2;
let <items>(<count>) = "Bread";
```

Assigning values

You can assign values to parameters or variables either as literals or as other variables. For example:

```
let <var1> = 2;
let <var2> = <var1>;
close <var2>;
close 3;
```

When coding text strings you must enclose the characters in double quotation marks and the text string itself must not span more than one line; for example:

```
let <var2> = "Text String";
```

To reference the contents of variables within a string you must prefix the reference with '%@' and enclose the token name in angle brackets as normal:

```
let <var3> = "Account: %@<var6>";
```

Control characters that form part of the script syntax itself must be treated as special cases when used within text strings. These are: quote mark: ‘ double quote: “ left and right angle brackets: < and > percent symbol: %. If you need to use any these characters within string assignments they must be repeated. For example:

```
let <var1> = "He said ""hello""";
let <var2> = "The variable is <<var2>>";
```

Alternatively, use strings enclosed in single quotes. These are always assumed to be fully literal – i.e. control characters are ignored and treated as text.

To assign a hex value you must use the X'nn,nn' format where nn is an individual hex reference. For example:

```
let <var1> = X‘F6,E8,F9,40’;
```

Calculation & concatenation

A calculation can be specified using the standard arithmetic operators: + (addition), * (multiplication), - (subtraction) and / (division). For example:

```
let <var1> = 6 * 10;
let <var2> = <var1> - 20;
let <var3> = <var1> + <var2>;
```

Similarly concatenation of two strings is specified using the + operator:

```
let <var1> = "Working" + " days";
let <var2> = <var1> + " in this month";
```

Compound statements are supported but you should only use them to assign a value to a variable and not as a direct part of other commands. For instance:

```
let <var1> = <var2> / (10 - <var3>);
let <var2> = "has " + <var1> + " days";
let <x> = (10 + <y>) * (3 + <y>);

let <st> = SUBSTRING ("abc" + "defghijk") (1 + 2) (7 - 1);
```

Comparing values

Comparing values results in a boolean TRUE/FALSE result which should be assigned to a variable.

Compare strings using the **equals** or **contains** functions.

Compare numbers by using these operators: `eq` – equal to `gt` – greater than `ge` – greater than or equal to `le` – less than or equal to `lt` – less than `ne` – not equal to. For example:

```
let <v> = 101 ge 102;   // <v> is FALSE
```

Existing boolean variables can be compared with each other using these logical operators `and` – two variables are TRUE `or` – either of two variables is TRUE `not` – a variable is not TRUE Additionally a variable can specifically be set to TRUE or FALSE as required. For example:

```
let <c1> = 6 gt 10;
let <c2> = "statement" contains "ate";
let <v> = <c1> or <c2>;   // <v> is TRUE
if <v>;
  let <test> = true;
else;
  let <test> = false;
end if;

let <bool> = <v> OR NOT (<c1> AND <c2>);
```

File Names

When you use the open command you will need to code file names in the format suitable for the platform on which the script is intended. File names must always be enclosed in double quotes.

z/OS

You can either specify fully qualified dataset names or reference datasets by Data Definition (DD) labels with actual files being assigned to these labels in the JCL use to start PCE. Note that PCE does not provide a mechanism for specifying dataset attributes so where open to a new output file is specified a DD reference should normally be used so that the relevant attributes can be specified in the JCL. Example:

```
open "DD:AFPDS1"...
open "USER1.PCE.INPUT"...
```

Windows

Files are referenced by path (optional) and filename. Example:

```
open "c:\doc\pceinput\bill.afp"...
```

UNIX

Under all supported UNIX systems, files are referenced by path (optional) and filename. Example:

```
open "/doc1host/pcein/afpds1.out"...
```

Comments

Comments that do not need to span a line are prefixed with a double slash and are assumed to occupy the remainder of the line. They can be used within a statement that spans lines. Examples.

```
//Start of totals calulation
let <var1> = <var2>; //Copy old value
let <var1> =     //Sum of sub-totals
var2 + <var3;
```

Comments that need to span lines must be prefixed with an asterisk. These must not appear within statements and are always assumed to occupy all of the final line. Examples:

```
let <string> = "Acc."; * This abbreviation is used throughout
* This procedure initializes variables at each pass
```

PCE command reference

Conventions used in syntax diagrams Parameters between braces {…} indicate that one should be selected from the available options which are separated by the ‘|’ delimiter. Optional parameters or groups of parameters are set between square brackets […].

add document id

Function

Adds (or changes) the PCE unique identifier that is assigned to the first page of documents intended for the Document Repository component.

Syntax

```
add document id of idString to firstPage ;
```

Parameters

idString text; the identifier to be stored in the page

firstPage variable or array element containing the first page of a document.

Effects

The identifier stored in idString will be inserted into the datastream page stored in firstPage. Any existing identifier will be overwritten

Comments

firstPage must be the first page of a document.

idString must contain a valid PCE identifier otherwise an error will be raised. The only method of getting a valid identifier string is to read one from an existing page using the **document id** command.

Example

```
...
//read first page from datastream containing the required id
read 1 page from file 1 into <oldAfpPage>;
//get the document id
let <DocId> = document id in <AfpPage>;
//Read in page to be manipulated
read 1 page from file 2 into <newAfpPage>;
//copy the id into the page
add document id of <DocId> into <newAfpPage>;
//write out amended page
write 1 page into file 3 from <newAfpPage>;
```

add document name

Function

Places pages within a group structure and adds (or changes) the name information for such a group.

Syntax

```
add document name of name to docGroup [at
{start|middle|end}];
```

Parameters

name text

docGroup variable containing/to contain a PCE document group.

Effects

name is used as the identifier for all group related datastream fields created by this command.

If the middle keyword is specified the pages in docGroup will be enclosed in the appropriate fields to form a complete document group.

If the start keyword is specified a field to indicate the start of a group will be inserted before the pages in docGroup.

If the end keyword is specified a field to indicate the end of a document group will be inserted after the pages in docGroup.

Comments

A document group can also be copied directly from input using the **read…document** command.

Typically the start and end keywords are used when you need to build a group from more pages than you want to hold in memory at one time. In this scenario you will need to code multiple write commands to build the group in the output file.

If docGroup already contains existing group name fields these will be overwritten with name.

If you want to use **TLE add** to add a group index reference to an AFP document you must ensure it is coded in the appropriate sequence. For instance, if you are using multiple read/writes to form the group the **TLE add** should be coded after the add document name…start but before the group is written to the output file.

This command is only valid when working with AFP pages. Using it with other datastreams will cause unpredictable results.

Example

```
//Creating a new group from a complete set of pages in
memory
read 10 pages from <InFile> into <Group>;
add document name of "Small Group" to <Group>;
add document TLE attrib "Small_Group" value "Index1" to
 <Group>;
write 1 item to <OutFile> from <Group>;

//Building a group using multiple read/writes
read 10 pages of document start from file<InFile> into
<Group>;
add document name of "Big Group" to <Group> at start;
TLE add at document of attrib "first_tle" value "Index1"
 to <Group>;
write 1 item to <OutFile> from <Group>;
read 10 pages of document middle from file<InFile> into
 <Group>;
write 1 item to <OutFile> from <Group>;
read 10 pages of document end from file <InFile> into
<Group>;
add document name of "Big Group" to <Group> at end;
write 1 item to <OutFile> from <Group>;
```

add medium map

Function

Inserts an AFPDS Invoke Medium Map structured field into an existing AFP page.

Syntax

```
add medium map mapName to afpPage ;
```

Parameters

mapName text; a string that forms a valid AFP medium map name

afpPage a variable or array element containing a single AFP page.

Effects

An AFPDS Invoke Medium Map structured field which references a medium map name of mapName will be inserted into the AFP page stored in afpPage.

Comments

A medium map is a functional definition of an AFP form definition and is also known as a copy group. Refer to your IBM AFP related documentation for more information on the uses of medium maps.

The medium map name used must be included in the form definition assigned to the print function when printing AFP manipulated by this command.

Example

```
...
open "DD:AFPIN" for input as file 1 record(8205)/AFPDS;
open "DD:AFPOUT" for output as file 2 record/AFPDS;
read 1 item from file 1 into <AFPPage>;
add medium map "MM1"> to <AFPPage>;
write 1 item into file 2 from <AFPPage>;
```

atrim

Function

Removes leading and trailing spaces from a text string.

Syntax

```
let result = atrim inputString ;
```

Parameters

result a variable to receive the resulting text string

inputString text; the string to be adjusted.

Effects

result is updated with the contents of inputString but with any leading and trailing spaces removed.

Example

```
...
let <var1> = " The quick brown fox ";
let <var2> = atrim <var1>; // <var2> = "The quick brown
 fox"
```

barcode

Function

Converts a string to a simple barcode format.

Syntax

```
let barcode = barcode
{PostNet|2of5|3of9|Code128A|Code128B|Code128C}
using input;
```

Parameters

barcode a variable to receive the formatted barcode string

input text; the string to be converted.

Effects

input is converted to a new string barcode. When presented using the appropriate barcode font the new string will produce the equivalent barcode image. Standard 'framing' characters will be applied to the start and end of the barcode string depending on the type specified.

Comments

You will need to use this command in conjunction with a composition edit (CE) sequence (**begin ce**/**end ce**) that associates the barcode string with the appropriate barcode font and includes it in an output datastream page.

IMPORTANT: this function is mainly provided for backward compatibility. A more comprehensive barcode formatting function is provided as the Do Barcode CE command. See **Composition Edit Commands** on page 172 for details.

Example

```
...
// Read page of AFP data
read 1 page from file 0 into <AFPDSPage>;
// Read barcode info from journal
read 1 items from file 1 into <JournalData>;
let <BarCode> = barcode 3of9 using <JournalData>;
  begin ce into <AFPDSPage>;
    =SCPP  001.500  002.500;;
    =STP  90;;
    =STL  X0BAR3O9 %@<BarCode>;;
  end ce;
```

begin ce

Function

Indicates the start of one or more composition edit (CE) commands which add new elements to an existing output datastream page.

Syntax

```
begin ce into Page;
```

Parameters

Page an output datastream page in any supported format stored in a variable. If the variable is currently empty then it is treated as a new page.

Effects

All text between the **begin ce** and **end ce** statements will be interpreted as commands in the format required for composition edit. The results of the CE commands will be merged into the page of output datastream stored in variable Page.

Comments

CE commands are evaluated separately from regular PCE script commands and have a different syntax. Commands are strictly order dependent and position sensitive so special care must be take when coding CE instructions. Refer to **“Composition Edit Commands” on page 157** for details of the syntax and detailed function of these commands.

Example

```
...
if <LastPage>;
  begin ce into <AFPPage>;
    =SCPP  001.500  002.500;;
    =STP  90;;
    =STL  X0T055A0 This is the last page of
%@<pagecount>;;
  end ce;
end if;
```

begin loop

Function

Indicates the start of one or more script statements that are to be processed a variable number of times.

Syntax

```
begin loop;
```

Effects

Statements between the begin loop and **end loop** statements will be repeatedly executed until an **exit loop** statement is actioned.

Comments

Loops can be nested but the user must ensure that the correct sequence of begin loop and **end loop** statements is maintained.

Example

```
declare <input>;
declare <n>;
declare <done>;
begin loop;
  read <n> items from file 1 into <input>;
  // Exit if less than 5 items read.
  let <done> = <n> ne 5;
  exit loop when <done>;
  ...
end loop;
```

begin procedure

Function

Indicates the start of one or more script statements that make up a procedure.

Syntax

```
begin procedure procName [as main];
```

Parameters

procName text; the procedure name enclosed in angle brackets.

Effects

All statements between this statement and the next **end procedure** statement are part of procedure procName. If as main is coded the procedure is treated as the main PCE process.

Comments

All procedures must be declared before use – see **“declare procedure” on page 81**. Note that if as main is not used in any procedure one of the procedures will be chosen at random from the script to be used as main.

Example

```
declare procedure <mainProc> as main;
declare procedure <initVars>;
begin procedure <mainProc>;
  ...
  call procedure <initVars>;
end procedure;
begin procedure <initVars>;
  let <DocTotal> = 0;
  let <RecordCounter> = 1;
end procedure;
```

call procedure

Function	Executes a procedure coded within the script.
Syntax	`call procedure procName ;`
Parameters	procName text; the procedure name enclosed in angle brackets.
Effects	The procedure with name procName is executed. On return from the procedure control passes to the command following this statement.
Comments	All procedures must be declared before use – see **declare procedure** on page 97. Procedures cannot be called recursively i.e. they cannot call themselves or a higher level procedure.
Example	See **begin procedure** on page 87.

call userexit

Function

Calls an external program via the user exit feature.

Syntax

```
[let result = ] call userexit id [with parameter in
cell...];
```

Parameters

result a variable to accept a return value

id text; indicates a user exit identifier

parameter a variable or array element containing text or a page of output datastream

cell integer; indicates a PCE memory cell.

Effects

The user defined program specified by the identifier id in the User Exit Control File assigned in the PCE initialization file is executed. The program can be passed one or more optional **parameters** values using the defined PCE memory cell.

If a value is returned by the user exit program will be stored in result if this is coded.

Comments

User exits of type PRINTSTREAM should return a self-contained segment of output datastream. This can be included within an existing page using the **insert object** on page 119 command.

Parameters can only contain a STRING or PAGE data types; NUMBER and DATE data types are not supported. The PCE memory cell for the parameter must be in the range 9900–9998 and must match the appropriate number specified in the Generate Engine Data Exchange API function being used with the user program. The result value, if used, will always be of type STRING.

Refer to **User Exits** for more information about the communication process between PCE and user programs, returning AFP image objects and coding the User Exit Control File.

Example

```
...
let <title> = "Customer_Name";
let <number> = "Customer_Number";
call userexit "WriteName" with <Title> in 9900;
let <uxResult> = call userexit "GetAddress" with <Number>
 in 9901;
let <parm> = substring<UXResult> 2 5;
```

change DIJelement

Function

Updates the value of an element of an entry in a Document Interchange Journal (DIJ).

Syntax

```
change DIJelement
{"Name"|"AddrLine1...AddrLine6"|"City"|"Region"|"PostalCode"|
"Country|Phone"|"NumberOfPages"|"SkippedPages"} value to
 value in dijRecord ;
```

Parameters

value text; the value with which the DIJ element is updated

dijRecord a variable containing a DIJ record.

Effects

The named element within the xmlRecord will be updated with the value contained in varValue. Any existing value is overwritten.

Comments

The DIJ record must have been read from a file opened as type DIJ.

There are 6 possible address line elements.

Example

```
declare <dijrecord>;
declare <zip>;
declare <newzip>;
// Open the journal
open "C:\gen\original\applic1.jrn" for input as file 1
line/DIJ;
open "C:\gen\new\applic1.jrn" for output as file 1
line/DIJ;
read 1 item from file 1 into <dijRecord>;
let <zip> = dijelement "PostalCode" in dijRecord;
let <newzip> = call userexit <datahygiene> with <zip> in
 9901;
let <zipchange> = <newzip> ne <zip>;
if <zipchange>;
   change DIJelement "PostalCode" value to <newzip> in
<dijRecord>;
end if;
write 1 item into file 2 from <dijRecord>;
```

close

Function

Explicitly closes a previously opened file during a PCE process.

Syntax

```
close {input|output} fileRef ;
```

Parameters

fileRef integer; the handle of any file opened by the current script.

Effects

The file previously opened with ID fileRef is immediately closed. Code input or output to reflect the function for which the file was opened – this is important in identifying the most efficient manner in which to close the file

Processing of the script continues. If an error occurs a return code from this function will be stored as system value sys_last_error. This can be queried as part of a user defined error routine via the **on error call** on page 134 function.

Comments

Open files are automatically closed at the end of any PCE process. In addition, an open file with an ID that is specified as part of a further open command will also be automatically closed.

Example

```
...
on error call <ioErrorRoutine>;
...
close input 1;
close output <fileCount>;
```

contains

Function

Searches a string for a substring.

Syntax

```
let test = stringToSearch contains stringToFind ;
```

Parameters

test variable to accept TRUE/FALSE

stringToSearch text; usually a variable

stringToFind text.

Effects

test is TRUE if stringToFind is contained within stringToSearch (or is the same as it).

Example

```
let <verify> = "dog, cat, bird" contains "cat";  //
<verify> is TRUE
```

convert resolution

Function

Changes all resolution settings within a page of AFP output.

Syntax

```
convert resolution in page afpPage from oldRes to newRes>;
```

Parameters

afpPage a variable or array element containing a single AFP page.

oldRes integer; the current resolution used in the page

newRes integer; the new resolution to be used in the page

Effects

Within the AFPDS structured fields that make up afpPage all parameters using a resolution of oldRes are converted to newRes.

Comments

Supported resolutions are 240, 300 and 1440

Example

```
declare <page>;
declare <n>;
open "in240.afp"  for input as file 1 wsafp/ afpds;
open "out300.afp" for output as file 2 wsafp/ afpds;

let <n> = 1;
read <n> pages from file 1 into <page>;
convert resolution in page <page> from 240 to 300;
write <n> pages into file 2 from <page>;
```

date

Function

Gets the current system date.

Syntax

```
let value = date [format formatCodes ];
```

Parameters

value. a variable to receive the time string

formatCodes text; string representing a date format

Effects

value is updated with a string representing the system date when the command is issued. formatCodes can be used to define the format of the date as in the following table. Note that the actual text used for date elements is customizable using <Preferences> sections within the PCE initialization file.

	Format code	Output	INI setting to customize
Day	d dd ddd dddd	1 – 31 01 – 31 Sun – Sat Sunday – Saturday	Day (first three characters) Day
Month	m mm mmm mmmm	1 – 12 01 – 12 Jan – Dec January – December	MonthAbbrev MonthName
Year	y yy yyy yyyy	00 – 99 00 – 99 19xx – 20xx 19xx – 20xx	

Format code	Output	INI setting to customize
\c	Inserts a character into the date string, e.g. \y places 'y' in the string	

Comments

You can change which initialization file <Preferences> section is currently active by using the **set preferences** on page 155 command.

Example

```
// Get and write date and time info
 to output
let <nowDate> = date format
"ddmmmyyyy";
let <nowTime> = time;
let <outString> = "Date: %@nowDate
 Time: %@nowTime";
write 1 item into file 0 from
<outString>;
```

day

Function

Gets a day name string from the active <Preferences> section of the PCE initialization file.

Syntax

```
let value = day keywordRef ;
```

Parameters

value. a variable to receive the preferences value

keywordNum integer; a reference to the relevant day name keyword.

Effects

value is updated with the value assigned to the keywordNumth 'Day' keyword in the currently active <Preferences> section of the PCE initialization file. For instance, if keywordNum = 3 then the value assigned to the Day3 keyword will be used.

Comments

You can change which initialization file <Preferences> section is currently active by using the **set preferences** on page 155 command.

Example

```
...
// Set which INI file preferences section to use
set preferences to 3;
 // Get and write the name of the 4th day
let <prefValue> = monthname 4;
write 1 item into file 0 from <prefValue>;
```

declare procedure

Function	Declares a procedure name.
Syntax	`declare procedure procName [as main];`
Parameters	procName text; the procedure name enclosed in angle brackets.
Effects	procName is reserved as the name of a procedure to be coded within the script or referenced by it. If as main is coded the procedure will be treated as the main PCE process.
Comments	The declared name cannot also be used as a variable name.
Example	See **begin procedure** on page 87

declare (variable)

Function

Declares a variable name.

Syntax

```
declare varName [(arrayElements)];
```

Parameters

varName text; the variable name enclosed in angle brackets

arrayElements integer; the number of elements contained by the variable.

Effects

varName is reserved as the name of a variable to be coded within the script or referenced by it. If arrayElements is coded the variable can be used as an array with arrayElements indicating the maximum number of elements.

Comments

If you are reading from a delimited input file into a variable – i.e. each record has multiple fields – the variable must be declared as an array and must have sufficient elements to cater for the number of fields in each record. This is typically a requirement for journal files.

Example

```
declare <count>;                //Loop counter
declare <JournalData>(5);       //Stores 5 journal fields
```

DIJelement

Function

Gets index elements from a record within a Document Interchange Journal (DIJ).

Syntax

```
let value = DIJelement
{"Name"|"AddrLine1...AddrLine6"|"City"|"Region"|"PostalCode"|
"Country|Phone"|"NumberOfPages"|"SkippedPages"} in
dijRecord;
```

Parameters

value a variable to receive the value read from the DIJ record

dijRecord a variable containing a DIJ record.

Effects

The named element within the DIJ record dijRecord is copied into value.

Comments

The file from which the record is taken must have been opened as type DIJ. There are 6 possible 'address line' elements.

Example

See **change DIJelement** on page 90.

docoffset

Function

Gets the file pointer of an output datastream file for the start of the last written document.

Syntax

```
let value = pageoffset;
```

Parameters

value. a variable to receive the offset number.

Effects

value is updated with the file pointer offset indicating the start of the last document that was written to an output datastream file. The offset value is the number of bytes from the start of the file. Note that the offset position ignores any document header data and marks the point where the first page within the document begins.

Comments

A document is written using the command. To qualify as a document pages must first be included in a document group variable. This can be achieved using the **read…document** on page 148 or **add document name** commands. Document groups are supported for AFP streams.

Use this command immediately after the write operation to the file for which it is intended to apply. You cannot request the offset of a specific file. This command is not supported for files opened for input.

Example

```
...
open <afpInFile> for input as file 1 wsafp/afpds;
open <afpOutFile> for output as file 2 wsafp/afpds;
// Read and write first page of AFP
read 10 pages of document start from file 1 into
<afpGroup>;
write 1 item to <afpOutFile> from <afpGroup>;
//Get last offset
let <docOffset> = docoffset;
```

document id

Function

Retrieves the unique Generate document identifier from the first page of a document.

Syntax

```
let id = document id in page ;
```

Parameters

id a variable to receive the identifier.

page variable or array element containing the first page of a document.

Effects

The Generate document identifier stored in page is copied to id.

Comments

Document identifiers are added to the datastream by Generate when a DIJ index is being created for the application – i.e. when the datastream is intended to be stored in the Document Repository component. The ID will always be stored in the first page of each document within the datastream.

If your PCE script has cause to remove or reorder the position of pages within a document you may need to use the **add document id** command to ensure that page 1 of each document has a valid identifier.

Example

See **add document id** on page 79.

document name

Function

Retrieves the name associated with a document group.

Syntax

```
let name = document name in docGroup;
```

Parameters

name a variable to receive the group name

docGroup a variable containing a PCE document group.

Effects

The reference name (if any) in the output datastream fields making up docGroup is copied into name.

Comments

To qualify as a document pages must first be included in a document group variable. This can be achieved using the **read…document** on page 148 or **add document name** commands. Document groups are supported for AFP streams.

Example

```
read 10 pages from <inFile> into <group>;
let <nameOfDoc> = document name in <group>;
```

document TLE

Function

Retrieves the AFP group level TLE information from a document group.

Syntax

```
let value = document TLE name in docGroup;
```

Parameters

value variable to receive the TLE attribute

name text; indicates the required attribute name

docGroup variable containing a document group of AFP pages.

Effects

The pages stored in docGroup are searched for an AFP group level index attribute name of name. If a match is found the data stored as the attribute value will be copied to value. If a match is not found value> will be a zero length string (which can be verified by a condition – see example).

Comments

This command searches for AFPDS Tag Logical Element (TLE) structured fields in docGroup. This is one of the structured fields that make up AFP indexing features.

This command is only valid when working with AFP datastream stored in a document group variable. Using it with other file formats will cause unpredictable results.

To qualify as a document pages must first be included in a document group variable. This can be achieved using the **read…document** on page 148 or **add document name** commands.

Example

```
let <firstgroupTLE> = document TLE <index1> in
<document3>;
let <OK> = Length <firstgroupTLE>;
if <OK>;
    write document into file <logfile> from <document3>;
else:
    trace "Index not matched";
end if;
```

end ce

Function	Indicates the end of PCE composition edit (CE) commands.
Syntax	`end ce;`
Effects	All statements following **end ce** on page 104 are treated as regular PCE statements.
Comments	See **begin ce** on page 85.
Example	See **begin ce** on page 85

end loop

Function	Indicates the end of statements within a loop.
Syntax	`end loop;`
Effects	Marks the end of loop statements started by **begin loop** on page 86.
Comments	Loops can be nested but the user must ensure that the correct sequence of **begin loop** on page 86 and **end loop** on page 105 statements is maintained.
Example	See **begin loop** on page 86.

end procedure

Function

Indicates the end of a PCE script procedure.

Syntax

```
end procedure;
```

Effects

This marks the end of the procedure indicated by the previous statement.

On encountering **end procedure** on page 106 control will be passed back to the command following the relevant **call procedure** on page 88 statement.

Comments

Note that if a **return** on page 152 statement is not encountered before an end procedure it is **begin procedure** on page 87 assumed.

Example

See **begin procedure** on page 87

equals

Function

Tests for equality of two strings.

Syntax

```
let test = str1 equals str2 ;
```

Parameters

`test` - variable to accept TRUE/FALSE

`str1` ,`str2` - text.

Effects

test is TRUE if str1 contains the same text as str2. This operation is case sensitive.

Example

```
let <verify> = "John Smith" equals "john smith"; //
<verify> is FALSE
```

exit loop

Function

Terminates loop processing when a condition is matched.

Syntax

```
exit loop when test ;
```

Parameters

test boolean variable; usually the result of a comparison statement – see **“Comparing values” on page 60** for details.

Effects

If test evaluates to TRUE the loop currently being processed is exited immediately. Processing restarts with the first statement after the current loop.

Example

```
...
let <count> = 0;
begin loop;
  let <count> = <count> + 1;
  // Exit loop if more than 5 executions.
  let <done> = <count> GT 5;
  exit loop when <done>;
  ...
end loop;
```

extract document page

Function

Extracts a page from a document group containing AFP pages.

Syntax

```
extract document page page in docGroup at
{start|end|pageNumber };
```

Parameters

page a variable that will receive the extracted page

docGroup a variable containing a document group of AFP pages

pageNumber integer; the sequence number of a page in docGroup.

Effects

A page of output datastream is extracted from docGroup and placed in page. If pageNumber is specified the page at the relevant position within the group is extracted. If start is specified the first page is extracted. If end is specified the last page is extracted.

Comments

If pageNumber does not indicate a valid page within the group no page is extracted.

The original page is completely removed from the group. You can reinsert it into a document group using the **move page** on page 129 command.

To qualify as a document pages must first be included in a document group variable. This can be achieved using the **read…document** on page 148 or **add document name** commands. Document groups are supported for AFP streams.

Example

```
// Open the I/O files (containing AFP data)
open "DD:PCEIN" for input as file 1 record(8205)/afpds;
open "DD:PCEOUT" for output as file 2 record(8205)/afpds;
//
// Read all the pages into the document group
let <pageCount> = 6;
read <pageCount> document from file 1 into <document2>;
...
// Get the page from the document group
extract document page <page1> in <document2> at start;
...
// Having worked on the page, move it back into the
document group
// & write the update document group back to file
move page <page1> to <document2> at start;
write document into file 2 from <document2>;
```

font

Function

Returns the reference information for a font in the active font list.

Syntax

```
let fontInfo = font fontNumber in {resources|datastream};
```

Parameters

`fontInfo` - an array variable of 6 elements to hold the returned information strings

`fontNumber` - integer; the sequence number of a font within the active font list.

Effects

If you specify the 'resources' keyword the active font list is determined from the fonts available in the HIP files specified in the PCE initialization file.

If you specify the 'datastream' keyword the active font list is set according to the font information that will be used when creating any output datastream from the current script. This will have been determined by theget resources command if this has been coded.

If you are adding text in PCE and using a font that is not in the input file, you must use the get resources from HIP command, ensuring that the HIP file contains the required font.

The font command returns an array of 6 strings:

1. Application name: the full font name as it is known to Generate; e.g. ":120.I2.400.Times New Roman.ANSI".

2. Device name: the name by which the font is referenced within the output datastream; e.g. “C0G00020:T1WF1148”.

3. Alternate device name: available for some datastreams or blank where not available; e.g. “X0G00020”

4. Encrypted name: interpretation of Application name; e.g. “TimesNewRoman-oblique.12”

5. Rotation: intended print direction (degrees) of font; will return “0”, “270” or “All”

6. Index: the reference number within the active font list. If this number has a positive value you may use it to reference the font within composition edit commands. If it has a negative value the font cannot be referenced in composition edit commands.

Comments

This is typically used to access available font names when adding text to existing pages (see **“Composition Edit Commands” on page 157** for details). Use it in conjunction with the **number of fonts** on page 131 command to determine how many times to iterate over the active font list.

Example

```
get resources from HIP;
let <maxFonts> = number of fonts in resources;
for <x> = 1 to <maxFonts>;
  let <fontInfo> = font <x> in resources;
  trace "Application Name '%@<fontInfo>(0)'";
  trace "Device Name '%@<fontInfo>(1)'";
  trace "Alternate Name '%@<fontInfo>(2)'";
  trace "Encrypt Name '%@<fontInfo>(3)'";
  trace "Rotation '%@<fontInfo>(4)'";
  trace "Index '%@<fontInfo>(5)'";
next;
```

for...next

Function

Indicates script statements that are to be processed repeatedly a fixed number of times.

Syntax

```
for [counter =] start to end ;
   ...
next;
```

Parameters

counter a variable to hold the loop counter

start integer; the base number of the loop counter when it is first processed

end integer; the number of the loop counter when processing is to end.

Effects

Statements between the for and next commands will be executed the number of times indicated by the difference between start and end. For example, `1 to 5` will execute the loop five times; `3 to 4` will execute the loop twice.

The current sequence number within the start/end range is used to update counter at the start of each pass.

Comments

If coded, the contents of the counter variable may be referenced within the loop. It can be omitted unless you want to nest **for...next** on page 112 loops in which case each must be assigned to a different variable.

Example

```
for <x> = 1 to 4;
                             // Get file name for <x>th
 output file
  // convert numeric <x> to string and append to base
file name
  let <xString> = STRING <x>;
  let <dataFileX> = <name> + <x>;

  // open <x>th file
  open <dataFileX> for output as file <x> LINE / PLAIN;

  // read fields from next input file record
  let <no-of-fields> = 7;
  read <no-of-fields> items from file 0 into <fieldArray>;

  // NOTE: arrays are indexed from 0
  let <no-of-fields> = <no-of-fields> - 1;
  for <n> = 0 to <no-of-fields>;

    // Output fields read from <x>th record of input file
    // as separate records in the <x>th output file

    write 1 item into file <x> from <fieldArray>(<n>);

  next; // <n>

next; // <x>
```

get resources

Function

Specifically defines the source for fonts and images to be included in output files.

Syntax

```
get resources from {HIP|input file fileNumber } [with
fontmap from input file fileNumber ];
```

Parameters

fileNumber is the number associated with a previously opened output datastream file.

Effects

The main keyword relates to the source from which resources to be used when creating output files will be gathered. If 'HIP' is specified the font and images will be gathered from the one or more HIP files as specified in the PCE INI file. If 'input file' is specified resources are gathered from the output datastream file indicated.

If the fontmap keyword is specified the file indicated will be used to provide the master list of fonts when merging multiple output datastream files.

Comments

Output datastreams created by the same PCE job will always reference a single set of resources. If you are working with multiple datastreams (and in particular if you intend to merge datastreams into a single file) use this command to define where PCE should look for resources to be used in new output files.

If this command is not coded PCE will use the resources stored in the first output datastream file to be read by a script. For this reason **get resources** on page 114 must be coded prior to any file read commands.

The fontmap is used when you are merging pages from multiple sources. Where possible assign the fontmap to a file that contains all the resources required in the resulting output. This will minimize the need to rebuild pages prior to output.

Example

```
declare <topOfPage>;
declare <mainPage>;
open "d:\gen\j1out.ps" for input as file 1
line/postscript;
open "d:\gen\j2out.ps" for input as file 2
line/postscript;
open "d:\gen\merge.ps" for output as file 3
line/postscript;
//Call get resources after file opens but before first
read
get resources from HIP with fontmap from input file 2;
read 1 items from file 1 into <topOfPage>;
read 1 items from file 2 into <mainPage>;
merge <topOfPage> into <mainPage>;
write 1 items into file 3 from <mainPage>;
```

if...else...end if

Function

Conditionally executes statements.

Syntax

```
if test;
  ...
  [else;
  ...]
end if;
```

Parameters

test boolean variable; usually the result of a comparison statement. See **"Comparing values" on page 60** for details.

Effects

Statements between the if statement and else or end if statements (whichever appears first) will be executed if test evaluates to TRUE.

If test evaluates to FALSE statements between the else and end if statements (if any) will be executed.

Comments

`if…else…end if` constructs may be nested but you must ensure that the statements are balanced.

Example

```
...
// Check current page count
let <check> = <count> lt 1;
// Over write text as appropriate
if <check>;
  overwrite "type to replace" in <StartPage> with
<HeadPage1>;
else;
  overwrite "type to replace" in <StartPage> with
<HeadPage2>;
end if;
```

include

Function

Inserts PCE script commands from another file into the current script.

Syntax

```
include file ;
```

Parameters

file text; the path/file name of a file containing a PCE script.

Effects

The contents of file are inserted into the calling script at the point this command is encountered.

Comments

The file to be inserted must contain PCE script code. The user must ensure the commands in the include file are in context at the point at which they inserted and that variable and procedure names are unique and consistent throughout. Include files usually contain code that can be reused by multiple PCE scripts, for example, common procedures.

Example

```
declare procedure <mainProc> as main;
declare procedure <copyfields>;

begin procedure <mainProc>;
  include "common_procs";
  call procedure <copyfields>;
end procedure <mainProc>;
```

in range

Function

Tests to see if an integer value is within a specified range.

Syntax

```
let test = queryValue in range lower..upper ;
```

Parameters

test variable to accept TRUE/FALSE

queryValue integer; the value to be tested

lower..upper an integer range, e.g. 1..40, -20..-1, 1000..2000, etc.

Effects

test is TRUE if num1 contains a value in the inclusive range represented by the values numLower and numUpper.

Example

```
let <verify> = 6 in range 4..9; // <verify> is TRUE
```

insert object

Function

Adds a self-contained segment of output datastream to an existing page.

Syntax

```
insert object {9999|object } at {start|end|objnum } of
page page;
```

Parameters

object a segment of output datastream valid with a user exit

objnum integer; indicates an object sequence within a page

page a variable containing a page of output datastream.

Effects

The contents of object or the userexit return cell 9999 are inserted into page. If start is specified it is inserted before existing data; if end is specified after existing data; if objnum is specified it is used to sequence the object with others already in the page.

Comments

The object to be inserted is normally provided by calling a user exit of type PRINTSTREAM. Only those objects supported by PRINTSTREAM user exits can be inserted using this command. Refer to the User Exits section of the DOC1 Suite 4 Programmer's Guide for details of the output datastreams supported and the required content of an object to be used in this context.

Example

```
...
read 1 page from file 0 into <AfpPage>;
let <Image> = "Image1";
let <UXResult> = call userexit <GetImage> with <Image>
in 9901;
insert object <UXResult> at start of page <currentpage;
```

length

Function	Returns the length of a string.
Syntax	`let result = length inputString ;`
Parameters	result a variable to receive the resulting value inputString text.
Effects	result is updated with the number of bytes making up inputString.
Example	`let <count> = length "Monthly report";    //count = 14`

ltrim

Function

Removes leading spaces from a text string.

Syntax

```
let result = ltrim inputString ;
```

Parameters

result a variable to receive the resulting text string

inputString text; the string to be adjusted.

Effects

result is updated with the contents of inputString but with any leading spaces removed.

Example

```
...
let <var1> = "      The quick brown fox";
let <var2> = ltrim <var1>;    // <var2> = "The quick
brown fox"
```

mapp

Function

Replaces lookup code sequences in a text string with values from a Generate lookup table file (also known as a text substitution file) using keys or sequence numbers.

Syntax

```
let result = mapp
"...@@{Nstr|DseqNum|Ivar|MprefNum}@@...";
```

Parameters

result a variable to receive the resulting text string

str text

seqNum integer; sequence number of an entry in the lookup table

var variable containing a text string or integer

prefNum integer; reference to a UserValue keyword in PCE preferences

Effects

Any text enclosed in double @ symbols is passed as a parameter to the function. This function performs a look-up of these parameters using the Generate lookup table (as assigned to the job using <Files>TextSubstution keywords in the PCE initialization file). Where a match is found the substitution value is used to replace the parameter in the original string. The new string with all substitutions made is stored in result. The look-up can be performed using a variety of methods as indicated by the first character within the @@ construct (which is ignored). These are as follows:

N – str is a label used in the lookup table.

D – seqNum is the sequence number of an entry in the lookup table where 1 is the first entry, 2 is the second and so on.

I – var is an indirect reference to an entry in the lookup table. If it contains a string it is assumed to be a label name. If it contains an integer it is assumed to be a sequence number.

M – prefNum is a literal reference to one of the 16 user value keywords in the currently active preference section of the initialization file. 1 = UserValue1, 2 = UserValue2 and so on. The value assigned to the keyword is used as the sequence number of an entry in the lookup table.

Comments

A lookup table file is normally a simple text file that can be created by any basic editor. The format of the file is as follows:

```
keyword   substitution string
keyword   substitution string
keyword   substitution string
...
```

Keywords must always be a single word without spaces or special characters and are case sensitive. Substitution strings can contains any number of words but must remain on a single line.

When EMFE is running under z/OS, lookup table data may optionally be stored as a Key Sequenced VSAM file (KSDS). For this format keywords must be entered as the VSAM key field and the associated substitution strings as the remaining record contents. Each KSDS record can contain only a single text substitution entry. In order to use this method you must code TextSubsMethod=VSAM in the PCE initialization file.

Example

```
//These examples assume the text subs file contains:
//day Wednesday
//month June
//type Quarterly Statement

let <var1> = "The month is @@Nmonth@@";
let <var2> = mapp <var1>;
//<var2> = "The month is June"

let <var1> = "The day is @@D1@@";
let <var2> = mapp <var1>;
//<var2> = "The day is Wednesday"

let <var1> = "period";
let <var2> = "This is your @@I<var1>@@";
let <var3> = mapp <var1>;
//<var3> = "This is your Quarterly Statement"

// INI file contains <Preferences> UserValue3 = 2
let <var1> = "The month is @@M3@@";
let <var2> = mapp <var1>;
//<var2> = "The month is June"
```

merge

Function

Merges two pages (or a page and object where supported) and optionally resizes.

Syntax

```
merge page1 into page2 [of new size xSize ySize inches]
 ;
```

Parameters

page1 a variable or array element containing a single page of output datastream or, if merging AFP, a BCOCA object

page2 a variable or array element containing a single page of output datastream

xSize number; the required page width specified in inches

ySize number; the required page height specified in inches.

Effects

All presentation objects in page1 are added to page2. Optionally, page2 is resized using the xSize and ySize parameters.

Comments

Both page1 and page2 must contain output datastream of the same type. If merging AFP page1 may contain a complete BCOCA object rather than a page.

Page width and height values are relative to the top left corner of the physical page as determined by the target output device. Consult your output device documentation for more information.

Example

```
...
open "DD:INPUT1" for input as file 1 record(8205)/AFPDS;
open "DD:INPUT2" for input as file 2 record(8205)/AFPDS;
open "DD:OUTPUT" for output as file 3 record(8205)/AFPDS;
...
read 1 items from file 1 into <topOfPage>;
read 1 items from file 2 into <mainPage>;
let <x> = 7.5;
let <y> = 11;
merge <topOfPage> into <mainPage> of new size <x> <y>
inches;
write 1 items into file 3 from <mainPage>;
```

mixc

Function

Applies mixed case conversion to a given string in conjunction with an exception dictionary.

Syntax

```
let result = mixc inputString;
```

Parameters

result a variable to receive the resulting text string

inputString text; the string to be adjusted.

Effects

If this function is used the PCE initialization file used must specify a Generate exception dictionary file. This function will search the exception dictionary for inputString ignoring all casing. If a match is found result will be updated with the string as it appears in the dictionary – i.e. the casing of inputString may be amended. If it is not found in the dictionary result will be updated with inputString but with 'standard' casing applied – i.e. the first character will be upper-case and all others lower-case.

Comments

The exception dictionary is a simple text file that can be created with any standard editor. It should simply contain one or more text strings with the required casing correctly specified. Each text string should appear on a separate line.

Example

```
...
//The dictionary contains "PhD"
let <var1> = mixc "phd"; // <var1> = "PhD"
//The dictionary does NOT contain "PhD" (using any casing)
let <var1> = mixc "pHD"; // <var1> = "Phd"
```

monthabbrev

Function

Gets a month name abbreviation string from the active <Preferences> section of the PCE initialization file.

Syntax

```
let value = monthabbrev keywordNum ;
```

Parameters

value. a variable to receive the preferences value

keywordNum integer; a reference to the relevant month abbreviation keyword.

Effects

value is updated with the value assigned to the keywordNumth 'MonthAbbrev' keyword in the currently active <Preferences> section of the PCE initialization file. For instance, if keywordNum = 3 then the value assigned to the MonthAbbrev3 keyword will be used.

Comments

You can change which initialization file <Preferences> section is currently active by using the **set preferences** on page 155 command.

Example

```
...
// Set which INI file preferences section to use
set preferences to 3;
 // Get and write the abbreviated name of the 9th month
let <prefValue> = monthabbrev 9;
write 1 item into file 0 from <prefValue>;
```

monthname

Function

Gets a month name string from the active <Preferences> section of the PCE initialization file.

Syntax

```
let value = monthname keywordRef ;
```

Parameters

value. a variable to receive the preferences value

keywordNum integer; a reference to the relevant month name keyword.

Effects

value is updated with the value assigned to the keywordNumth 'MonthName' keyword in the currently active <Preferences> section of the PCE initialization file. For instance, if keywordNum = 3 then the value assigned to the MonthName3 keyword will be used.

Comments

You can change which initialization file <Preferences> section is currently active by using the **set preferences** on page 155 command.

Example

```
...
// Set which INI file preferences section to use
set preferences to 3;
 // Get and write the name of the 9th month
let <prefValue> = monthname 9;
write 1 item into file 0 from <prefValue>;
```

move

Function

Moves all presentation elements in a page.

Syntax

```
move page by xOffset yOffset inches;
```

Parameters

page a variable or array element containing a single page of output datastream

xOffset number; the amount to move vertically specified in inches

yOffset number; the amount to move horizontally specified in inches.

Effects

All presentation objects within page are moved vertically by xOffset and horizontally by yOffset.

Comments

Either or both offset values may be negative if required.

It is the user's responsibility to ensure that all presentation elements still fall within the boundaries of the logical page following a move. Results will be unpredictable where this is not the case.

Example

```
...
open "DD:INPUT1" for input as file 1 record(8205)/AFPDS;
open "DD:OUTPUT" for output as file 2 record(8205)/AFPDS;
read 1 items from file 1 into <afpPage>;
let <x> = 1.5;
let <y> = -1;
move <afpPage> by <x> <y>  inches;
write 1 items into file 2 from <afpPage>;
```

move page

Function	Moves a page into a document group
Syntax	`move page page to docGroup at {start|end|pageNumber };`
Parameters	page a variable or array element containing a single page of output datastream docGroup a variable containing a PCE document group pageNumber integer; the sequence number of an existing page in docGroup.
Effects	page is copied into docGroup. If start is specified the page is added as the first page in docGroup. If end is specified the page is added as the last page in docGroup. If pageNumber is specified page is added to docGroup following the page number indicated. If pageNumber references a non-existent page, page is inserted at the end of docGroup After the move the page variable is emptied.
Comments	To qualify as a document pages must first be included in a document group variable. This can be achieved using the **read…document** on page 148 or **add document name** commands. Document groups are supported for AFP streams.
Example	See **extract document page** on page 109.

nop

Function

Gets the specified NOP comment from an AFP page.

Syntax

```
let value = nop in afpPage at {start|end|sequenceNum };
```

Parameters

value. a variable to receive the NOP string

afpPage a variable or array element containing a single AFP page.

sequenceNum integer.

Effects

value is updated with the comment text from an AFPDS NOP (no operation) structured field within afpPage. If start is specified the comment is read from the first such structured field. If end is specified the last NOP field is used. If num is specified the comment is read from the sequenceNumth NOP field. If the specified NOP is not found no value is returned.

Comments

This command is only valid when working with an AFP page. Using it with other output datastreams will cause unpredictable results.

Example

```
// pick up 2nd comment
let <comment2> = nop in <AFPpage> at 2;
```

number of fonts

Function	Returns the number of fonts included in the file header of an input file.
Syntax	`let fCount = number of fonts in {resources|datastream};`
Parameters	fCount a variable to hold the font count
Effects	If you specify the' resources' keyword the count is determined from the fonts available in the HIP files specified in the PCE initialization file. If you specify the 'datastream' keyword the count is set according to the font information that will be used when creating any output datastream from the current script. This will have been determined by theget resources command if this has been coded.
Comments	Use this command to determine how many times to iterate over the active font list.This is typically used in conjunction with the **font** on page 110 command to access available font names when adding text to existing pages (see **Composition Edit Commands** on page 172 for details).
Example	See **font** on page 110.

numericconvcode

numericconvcode	
Function	Gets the string assigned to the NumericConvCode keyword from the active <Preferences> section of the PCE initialization file.
Syntax	`let value = numericconvcode;`
Parameters	value. a variable to receive the preferences value
Effects	value is updated with the value assigned to the 'NumericConvCode' keyword in the currently active <Preferences> section of the PCE initialization file.
Comments	NumericConvCode is used to specify the regional equivalent of the characters: .(period) ,(comma) +(plus sign) -(minus sign) and _(space) when formatting numeric data. You can change which initialization file <Preferences> section is currently active by using the **set preferences** on page 155 command.

Example	`...`
	`// Set which INI file preferences section to use`
	`set preferences to 3;`
	` // Get the numeric conversion codes`
	`let <prefValue> = numericconvcode;`

numericpadding

Function

Gets the character assigned to the NumericPadding keyword from the active <Preferences> section of the PCE initialization file.

Syntax

```
let value = numericpadding;
```

Parameters

value. a variable to receive the preferences value.

Effects

value is updated with the value assigned to the 'NumericPadding' keyword in the currently active <Preferences> section of the PCE initialization file.

Comments

You can change which initialization file <Preferences> section is currently active by using the **set preferences** on page 155 command.

Example

```
...
// Set which INI file preferences section to use
set preferences to 3;
 // Get the numeric padding character
let <prefValue> = numericpadding;
```

on error call

Function

Specifies a procedure to be called in the event of an IO error.

Syntax

```
on error call procedureName ;
```

Parameters

procedureName the name of a procedure available to the script.

Effects

Once this command has been processed within a script any subsequent **open** on page 136, **close** on page 91, **read** on page 145, **write** on page 169, or **overwrite** on page 141 commands that return an error will cause procedureName to be invoked.

Comments

Only one procedure can be called in this manner during a PCE process. The command can be specified anywhere in the script but only becomes active once it has been processed. Once activated you cannot return to the default functionality for PCE IO errors.

The procedure acting as the error routine can query the nature of the error via the sys_last_error system value and determine the appropriate action.

```
sys_last_error    Meaning
       21            The open command could not open an input file
       22            The open command could not open an output file
       23            The close command was unable to close an input file
       24            The close command was unable to close an output file
       25            A replace or overwrite command has failed
       33            An invalid file ID has been specified
       95           A non-existent file has been specified in a read or write command

       126           The format of a read or write command is incorrect
       129           Unable to write to an output file
```

Note that if the procedure does not specifically issue the **quit** on page 144 command PCE will continue with the script starting with the next statement after the command that caused the error.

Before on error call is processed PCE terminates and issues a generic error message if an IO error is encountered.

Example

```
declare procedure <MainProc> as main;
declare procedure <ErrorRoutine>;
on error call <ErrorRoutine>

begin procedure <MainProc>;
...
end procedure <MainProc>;

begin procedure <ErrorRoutine>;
  // Evaluate error codes and take appropriate action
  let <ErrorCheck> = <sys_last_error> eq 22;
  if <ErrorCheck>;
    trace "Output file not opened";
    let <sys_exit_value> = 12;
    quit;
  end if;
  let <ErrorCheck> = <sys_last_error> eq 24;
  if <ErrorCheck>;
    trace "Unable to close output file";
  end if;
  let <ErrorCheck> = <sys_last_error> eq 129;
  if <ErrorCheck>;
    trace "Unable to write to output file";
  end if;
end procedure;
```

open

Function

Declares and opens a file for processing.

Syntax

```
open fileName for {input|output|append} [using table
tableId] as file fileNumber [recordType[/fileType]];
```

Parameters

fileName the path/file name of the file to be opened. Refer to **"File Names" on page 61** for details of how files are identified on the various supported operating systems

tableId integer; the ID of a table in the assigned translation tables file.

fileNumber integer; the handle by which the file is to be referenced in other commands. All such handles must be unique within a PCE script.

recordType text; indicates the record construct of the file. See below.

fileType text; indicates what type of data is in the file. See below.

tableId

This option may be required if you are working with PostScript data that is not encoded in ASCII format or a regular text file that was created on a different operating system. A Generate translation tables file may be assigned to a PCE job as part of the Initialization File. tableid indicates the ID of a specific table that will be used to interpret text data where appropriate.

recordType

If this parameter is not specified the default settings are: Windows and UNIX: line z/OS.

If the default is not acceptable specify one of the following settings:

`line` A carriage return/line feed (CR/LF) at the end of each record. Use this setting for most file types when processing on Windows or UNIX where wsafp and wsmeta do not apply. Note: if using ASCII Postscript on EBCDIC platforms you need to code format(line).

`record [(numRecSize )]` The standard record format used for z/OS; an unblocked file with variable length records. numRecSize optionally specifies the record length associated with the file. If you are writing to a variable blocked dataset under z/OS you should specify a value greater than the longest possible record length to be produced (e.g. 8205 for AFPDS). For fixed blocked datasets specify the block size or greater. The **read** on page 145 functions will cease at end of record regardless of length specified.

When Reading and writing EBCDIC Postscript using DOC1PCE on Z/OS , specify the equivalent DOC1GEN (internal) Format Option when opening Postscript files for read or write:

```
// Open DOC1GEN EBCDIC PostScript Output For Reading as
Input  by DOC1PCE
OPEN "DD:PCEIN" FOR INPUT AS FILE 1
FORMAT($BN($RV($CC,$PD,$HV(0d,25)))) / Postscript;

                              // Open DOC1PCE EBCDIC
PostScript Output For Writing by DOC1PCE
OPEN "DD:PCEOUT1" FOR OUTPUT AS FILE 1
FORMAT($BN($RV($CC,$PD,$HV(0d,25)))) / Postscript;
OPEN "DD:PCEOUT2" FOR OUTPUT AS FILE 2
FORMAT($BN($RV($CC,$PD,$HV(0d,25)))) / Postscript;
```

`wsafp` A predefined construct suitable for AFP under all platforms other than z/OS. If you were using the format keyword this would equate to the parameter sequence: `$BN($RV($RD))`

`vsamafp / ksdsafp / ksdsmtc [(numRecSize )]` Predefined constructs suitable for output datastream stored as a VSAM dataset under and z/OS.

- vsamafp – AFP in a VSAM relative record dataset (RRDS)
- ksdsafp – AFP in a VSAM keyed sequence dataset (KSDS)

The dataset has fixed length records of size numRecSize. If you do not specify a length parameter PCE will assume a record length of 100. Notes: KSDS must use a key length of 10 bytes starting in the first byte of the record. An output datastream page always starts at the beginning of a record 'slot' but may span multiple slots. The final slot occupied by a page is padded to the start of the next slot.

`format(strFormatParms)` Use this setting if the file does not have a record construct catered for by the predefined keywords. Where this is the case you will need to supply details of the format as strFormatParms. Full details of the options for this parameter can be found in **Output datastream formats** on page 336.

fileType

Unless the default setting of plain is acceptable, specify one of the following settings. Do not forget to code the slash character before this parameter even if you are accepting the default for recordType:

`plain` This is the default on all platforms and indicates the file is not delimited by any of the methods indicated below. Each read/write operation will get/put an individual record as determined by recordType. Use this setting when reading most text files including journal files other than AFP or DIJ types.

	`delimited ({SepCharacter |SepNumber} [F,K,E])` The file contains two or more fields in each record and the fields are separated by the character specified. The field separation character may be identified as a literal text character or as a number representing the character in the standard code page for the relevant operating system. Each read/write operation will get/put a single field from the file. By default, PCE will ignore any field that is empty (contains zero bytes). The following options allow you to specify different behaviour: F – skip first field if empty, keep remaining empty fields as a NULL string K – keep all empty fields as a NULL string Where your read command specifies more fields than exist in a single record PCE will continue reading fields from the subsequent record by default. Where such overflows occur any remaining data in the subsequent record is ignored. You can specify the following option to control this: E – a read always stops at end of record regardless of the number of items specified. Subsequent reads commence at the next record.
	`afpds` The file contains AFP datastream. Each read/write operation will get/put the AFPDS structured fields that make up a single composed AFP page.
	`postscript` The file contains PostScript datastream. Each read/write operation will get/put the data structures that make up a single composed PostScript page.
	`line` Records are delimited by carriage return/line feed (CR/LF). Each read/write operation will get/put one such a record.
	`DIJ` The file contains a Document Interchange Journal. This is an XML construct which passes document indexes to the Vault environment. Each read/write operation will get/put the index entry related to a single document.
Effects	PCE attempts to open fileName for processing in the specified mode and, if successful, assigns a handle of fileNumber by which it is referred to in subsequent PCE commands. If input is specified PCE looks for an existing file with fileName and, if found, opens it for reading. If output is specified PCE attempts to create a new file with fileName and to open it for writing. An existing file with the same name may be overwritten if permitted by the operating system. If append is specified PCE looks for an existing file with fileName and, if found, opens it for writing and positions the file pointer at the end of the existing data. Note that append is not supported for files holding output datastream. The recordType and FileType attributes are associated with the file. Together these instruct PCE how to read/write a file and what amount of data is required for each read/write. If an error occurs a return code from this function will be stored as system value <sys_last_error> which can be interrogated using the **on error call** on page 134 procedure.
Comments	If the attribute parameters are not coded the defaults for UNIX and NT are "line/plain" and for z/OS "record/plain".

Example

```
...
//AFP under OS/390 or z/OS using a JCL DD card reference
open "DD:AFPIN" for input as file 0 record(8205)/afpds;
//PostScript under OS/390 or z/OS using a direct PDS reference
open "USR1.PS(J1)" for output as file 1 record(300)/postscript;
//AFP under Windows
open "d:\gen\pcein\j1.afp" for input as file 3 wsafp/afpds;
//PostScript under UNIX
open "/gen/j1out.ps" for output as file 4 line/postscript;
//space delimited text journal under OS/390 or z/OS
open "USR1.JRN(J1)" for append as file 6 record(80)/delimited(64);
//space delimited text journal under Windows
open "c:\gen\j1.jrn" for output as file 7 line/delimited(" ");
//Text file with non standard record format under Windows
open "d1.txt" for input as file 8 format($BN($RV($HV(‘0A’))))/plain;
```

ordinal

Function

Gets an ordinal value from the active <Preferences> section of the PCE initialization file.

Syntax

```
let value = ordinal keywordNum ;
```

Parameters

value. a variable to receive the preferences value

keywordNum integer; a reference to the relevant ordinal keyword.

Effects

value is updated with the value assigned to the keywordNumth 'Ordinal' keyword in the currently active <Preferences> section of the PCE initialization file. For instance, if keywordNum = 3 then the value assigned to the Ordinal3 keyword will be used.

Comments

You can change which initialization file <Preferences> section is currently active by using the **set preferences** on page 155 command.

Example

```
...
// Set which INI file preferences section to use
set preferences to 3;
 // Get and write the ordinal represnting number 9
let <prefValue> = ordinal 9;
write 1 item into file 0 from <prefValue>;
```

overwrite

Function

Finds and replaces a text string within a page without affecting file offsets.

Syntax

```
overwrite oldString in page with newString [once];
```

Parameters

oldString text; the string to be replaced

page a variable or array element containing a single output datastream page.

newString text; the string to replace oldString.

Effects

One or more occurrences of oldString within page are overwritten with the newString. Regardless of the content of either string the number of bytes occupied by the replacement string will always be identical to that occupied by the original. Truncation may occur or some bytes may remain unchanged if the two strings are of different length.

If the optional parameter once is specified only the first occurrence of oldString will be overwritten – subsequent occurrences will not be affected.

If an error occurs, for example if the search string is not found or if the write is unsuccessful, a return code from this function will be stored as system value <sys_last_error> which can be interrogated using the **on error call** on page 134 procedure.

Comments

The string parameters of the overwrite function are case sensitive.

If you need to replace strings of different length without truncation or padding use the **replace** on page 151 statement.

Specifying once will improve the performance of this command in most cases.

Example

```
...
if <page1>;
  let <overwriteText> = "Page 2 is next";
else;
  let <overwriteText> = "Page 2 is here";
end if;
overwrite "**replace_me**" in <afpPage> with
<OverwriteText>;
on error call <ErrorProc>;
```

page count

Function

Gets the number of pages in a document group.

Syntax

```
let pageCount = page count of docGroup;
```

Parameters

pageCount a variable to receive the page count value

docGroup a variable containing a PCE document group.

Effects

The number of pages stored in the PCE document group docGroup is stored in pageCount.

Comments

To qualify as a document pages must first be included in a document group variable. This can be achieved using the **read…document** on page 148 or **add document name** commands. Document groups are supported for AFP streams.

Example

```
read 10 pages from <inFile> into <group>;
let <numberOfPages> = page count of <group>;
```

pageoffset

Function

Gets the file pointer of an output datastream file for the start of the last written page.

Syntax

```
let value = pageoffset;
```

Parameters

value. a variable to receive the offset number.

Effects

value is updated with the file pointer offset indicating the start of the last page that was written to an output datastream file. The offset value is the number of bytes from the start of the file.

Comments

Use this command immediately after the write operation for the file to which it is intended to apply. You cannot request the offset of a specific file. This command is not supported for files opened for input.

Example

```
...
open <outputList> for output as file 0 line/plain;
open <afpInFile> for input as file 1 wsafp/afpds;
open <afpOutFile> for output as file 2 wsafp/afpds;
// Read and write first page of AFP
read <readNumber> page from file 1 into <AFPData>;
write 1 page into file 2 from <AFPData>;
//Get and write document offset to output
let <pageOffset> = pageoffset;
let <offsetOut> = string <Offset> 10 zero;
write 1 item into file 1 from <OffsetOut>;
```

quit

Function

Immediately terminates processing.

Syntax

```
quit;
```

Effects

On encountering this statement the PCE program will terminate immediately. No further statements in the script will be processed

Comments

All open files will be closed automatically.

Example

```
...
// Check that a full read took place...
let <BadRead> = <ReadNumber> lt 5;
// ... if not abort program
if <BadRead>;
  quit;
end if;
```

read

Function

Reads one or more items from a file into a variable.

Syntax

```
read [at offset] count {items|pages} from file fileRef
into var;
```

Parameters

offset integer; indicates the point at which reading should start (see below)

count integer; number of pages or other units of data to read

fileRef integer; handle of a file opened for reading

var a variable or array to receive the data read.

Effects

count items are read from the file referenced by fileRef and stored in var. If offset is specified reading begins at the file offset indicated otherwise reading begins using the file pointer as it was last updated.

If var is not an array and more than 1 item is specified the items read will be concatenated together in var. If var has been declared as an array and more than one item is specified each item will be read into a separate array element.

The amount of data read for each item depends on the file type specified when the file it was **open** on page 136ed. For files opened as output datastreams PCE always reads a page for each item specified. For other file types a single record is typically read for each item specified.

offset has two different bases depending on the platform running the script. z/OS offset represents a record count from the start of file. The first record is considered to be 1. (Note: on z/OS this value can be used with the VSAM files as well as other file formats.)

Windows & UNIX offset is a byte offset from the start of file. The first byte is considered to be byte 0 (zero). In both cases, if the data for offset is being read from a journal file the required values can be generated by using a vector offset environment variable as part of a journal entry in the document design. Refer to the Designer User's Guide for more information.

Following a successful read, the file pointer of fileRef will be updated to the position immediately after the data read.

Reading will stop before the number of specified items if end of file is reached or if var is an array but has less array elements than the number of items specified. If count is a variable it will be updated to reflect the number of items actually read. If required, the variable can be queried to ascertain how many pages have been read.

If an error occurs a return code from this function will be stored as system value <sys_last_error> which can be interrogated using the **on error call** on page 134 procedure.

Comments

If you use this command to read pages from an existing document group any fields in the datastream that control the group will be discarded. Consider using **read…document** on page 148 if required.

Example

```
Declare <JournalData>(5);
Declare <AFPData>;
...
let <ReadNumber> = 2;
// Read from journal
read <ReadNumber> items from file 0 into <JournalData>;
// Check that a full read took place...
let <BadRead> = <ReadNumber> lt 2;
// ... if not abort program
if <BadRead>;
  trace "Insufficient data in journal file";
  quit;
end if;
// If the journal entry shows the type required...
let <MatchedAccount> = <JournalData>(0) equals "Type 2";
//...read the AFPDS page at the appropriate file offset
if <MatchedAccount>;
  read at <JournalData>(1) 1 page from file <AFPFile>
into <AFPData>;
end if;
```

read...DIJentry

Function

Reads an entry from a Document Interchange Journal.

Syntax

```
read count DIJentry from file fileRef into var;
```

Parameters

count integer; number of DIJ entries to be read. Must be 1.

fileRef integer; handle of a DIJ file opened for reading

var a variable or array to receive the data read.

Effects

var is updated with the next entry in the DIJ file referenced by fileRef. Following a successful read, the file pointer will be moved to the start of the subsequent entry so that further reads can take place.

Comments

The file to be read must have been **open** on page 136 as type DIJ.

In the current version of PCE you may only read one entry at a time, i.e. count must be the constant '1' or a variable containing 1.

Refer to **DIJelement** on page 99 for information about reading specific elements from a DIJ entry.

Example

See **change DIJelement** on page 90.

read...document

Function

Reads pages into a document group.

Syntax

```
read [at offset] count pages of document
[{start|middle|end|all}] from file fileRef into docGroup
 ;
```

Parameters

offset integer; normally a vector offset value read from a journal

count integer; number of pages to read

fileRef integer; handle of an open output datastream file

docGroup variable to contain the resulting document group.

Effects

count pages are read from the output datastream file referenced by fileRef into docGroup. If offset is specified reading begins at the file offset indicated otherwise reading begins using the file pointer as it was last updated.

If the read produces a complete document group (as indicated by group fields within the output datastream) this will be copied without modification to docGroup. The group related fields are automatically retained along with the pages.

If the pages were not previously a complete document but the all keyword is specified a new document group will be created within the variable. This will be built using group fields suitable to the type of output datastream being read.

If the start, middle or end keywords are specified it is assumed that a partial read of an existing group is being performed. The keyword indicates which part of the group structure is expected and hence which group related fields from the datastream should be retained. If you are using this method you must code commands in the appropriate sequence to maintain the integrity of the group.

offset has two different bases depending on the platform running the script. z/OS offset represents a record count from the start of file. The first record is considered to be 1. (Note: on z/OS this value can be used with the VSAM files as well as other file formats.)

Windows & UNIX offset is a byte offset from the start of file. The first byte is considered to be byte 0 (zero).

In both cases, if the data for offset is being read from a journal file the required values can be generated by using a vector offset environment variable as part of a journal entry in the document design. Refer to the Designer User's Guide for more information.

If an error occurs a return code from this function will be stored as system value <sys_last_error> which can be interrogated using the **on error call** on page 134 procedure

Comments

This command is only supported when reading AFP output datastreams. Using it with other datastreams will cause unpredictable results.

docGroup does not need to be declared as an array as all the pages read are stored in a single element along with any existing group fields.

If you use a regular **read** on page 145 command to populate a document group the relevant group related fields will not be included in the variable.

Example

See **add document name** on page 80.

release

Function

Releases memory used by the data in a variable.

Syntax

```
release variable ;
```

Parameters

variable name of a declared variable.

Effects

Any data stored in variable is cleared and the system memory assigned to it is released. The variable remains available and retains its properties as assigned in the declare command.

Comments

Care must be taken not to read from the variable until data has been read into it.

Example

```
...
declare <DocData>;    // Holds  groups of records.
read 10 document from file 1 into <DocData>;
write 1 item into file 3 from <DocData>;
//release the memory used by <DocData>;
release <DocData>;
//reuse the variable for new data
read 1 document from file 1 into <DocData>;
```

replace

Function

Finds and replaces a string in a page. String length may be changed.

Syntax

```
replace oldString in page with newString [once];
```

Parameters

oldString text; the string to be replaced

page a variable or array element containing a single output datastream page.

newString text; the string to replace oldString.

Effects

One or more occurrences of oldString within page are overwritten with the newString. If the space required to stored newString is different than that required for oldString the remaining data within page is adjusted to compensate.

If the optional parameter once is specified only the first occurrence of oldString will be overwritten – subsequent occurrences will not be affected.

If an error occurs, for example if the search string is not found or if the write is unsuccessful, a return code from this function will be stored as system value <sys_last_error> which can be interrogated using the **on error call** on page 134 procedure.

Comments

If you need to replace strings without affecting length use the **overwrite** on page 141 statement.

The string parameters of the replace function are case sensitive.

If performance is an issue, use **overwrite** on page 141 rather than this command if possible. Specifying once will improve the performance in most cases.

Example

```
...
if <page1>;
  let <replaceText> = "Page 1 needs a large text string
 here";
else;
  let <replaceText> = "Others do not";
end if;
replace "***replaces***" in <mtcPage> with <replaceText>;
on error call <ErrorProc>;
```

return

Function	Causes an immediate return from a procedure.
Syntax	`return;`
Effects	On encountering this statement processing of the current procedure will stop and control will be passed back to the statement following that which called the procedure.
Comments	A **return** on page 152 is implied by the **end procedure** on page 106 statement. There is no need to code the actual statement in this circumstance.
Example	See **declare procedure** on page 97.

rtrim

Function

Removes trailing spaces from a text string.

Syntax

```
let result = rtrim inputString ;
```

Parameters

result a variable to receive the resulting text string

inputString text; the string to be adjusted.

Effects

result is updated with the contents of inputString but with any trailing spaces removed.

Example

```
...
let <var1> = "The quick brown fox      ";
let <var2> = rtrim <var1>;    // <var2> = "The quick
brown fox"
```

set page name

Function

Defines a page name to be included in PostScript output.

Syntax

```
set page name to pageName;
```

Parameters

pageName text; the name to be given to PostScript pages.

Effects

pageName is added as part of a "%%Page:" DSC comment whenever a PostScript page is written to an output file. Once this command has been processed it remains in force until a further **set page name** on page 154 command is encountered.

Comments

Each PostScript page generated by Generate starts with a %%Page DSC comment (Document Structuring Conventions). Such commands provide information about the PostScript file to a document manager or intelligent printer. %%Page has two arguments: Page Name – can be any value. This will be updated with pageName. Page Position – the position of the page within the PostScript file. This is automatically set.

If this command is never encountered in the script existing DSC Page comments are written unchanged to output files.

Example

```
open "\gen\transfers.ps" for input as file 2
line/postscript;
open "\gen\amended.ps" for output as file 3
line/postscript;
...
begin loop;
    // extract the customer account number...
    read 1 items from file 2 into <cust_account_no>;
   // ...and write it into the DSC comment for PostScript

    set page name to <cust_account_no>;
    write 1 items into file 3 from <cust_account_no>;
end loop;
```

set preferences

Function

Selects which INI file preferences section is active.

Syntax

```
set preferences to {prefNumber | default};
```

Parameters

prefNumber integer; reference to the <Preferences> section to be activated.

Effects

If prefNumber is coded the <Preferences> section in the PCE initialization file (INI) it indicates is activated. For instance, if prefNumber = 3 then <Preferences3> becomes the active section.

If default is coded <Preferences0> becomes the active section or, if this is not coded in the INI file, system defaults for all preferences settlings are assumed. This will also apply if prefNumber indicates a section that is not coded in the INI file.

Comments

The active preferences settings influence the values returned by the following commands: **day** on page 96, **monthname** on page 127, **monthabbrev** on page 126, **numericconvcode** on page 132, **numericpadding** on page 133, **ordinal** on page 140 and **uservalue** on page 167

Example

```
...
set preferences to 3;
let <prefValue> = uservalue 7;
```

string

Function

Converts a number to a formatted text string.

Syntax

```
let result = string inputNum format
[{left|right|zero|vector}];
```

Parameters

result a variable to receive the resulting text string

inputNum number; the value to be converted

format number; indicates the number of characters available when formating the text string.

Effects

result is updated with the result of converting inputNum to a text representation of a decimal number formatted within the number of characters indicated. format can have the following styles: 0 – (zero) use the minimum number of characters required to represent inputNum. n – represent inputNum as an integer only with n characters (padded as necessary) n.d – represent inputNum as a decimal number with n characters for the integral part and d characters for the fractional part. Both parts are padded as necessary. The decimal point will be included in the output string. Note that the minus sign is considered as part of the integral count if a negative number is specified

If format is other than zero the remaining parameters indicate how to justify the text within the available character spaces. Left – pad with spaces to the left of the string Right – pad with spaces to the right of the string Zero – pad with zeroes Vector – format in the style required for a journal vector offset entry. If this option is used format must be specified as 10.

Example

```
...
let <var2> = 3;
let <var1> = string <var2> 0 left;          //<var1> = "3"
let <var1> = string <var2> 2.2 zero;        //<var1> = "03.00"
let <var1> = string <var2> 4.2 right;       //<var1> = "   3. "
let <var2> = 3.5;
let <var1> = string <var2> 6 left;          //<var1> = "3     "
let <var1> = string <var2> 6 right;         //<var1> = "     3"
let <var1> = string <var2> 6 zero;          //<var1> = "000003"
let <var1> = string <var2> 4.2 right;       //<var1> = "  3.50"
let <var1> = string <var2> 4.2 zero;        //<var1> = "003.50"
let <var1> = string <var2> 1.5 zero;        //<var1> = "3.5000"
let <var2> = -3.5;
let <var1> = string <var2> 6 left;          //<var1> = "-3    "
let <var1> = string <var2> 6 right;         //<var1> = "    -3"
let <var1> = string <var2> 6 zero;          //<var1> = "0000-3"
let <var1> = string <var2> 4.2 right;       //<var1> = " -3.50"
let <var1> = string <var2> 4.2 zero;        //<var1> = "0-3.50"
let <var1> = string <var2> 2.4 zero;        //<var1> = "-3.500"
```

substring

Function

Extracts a substring from a text string.

Syntax

```
let result = substring inputString start length ;
```

Parameters

result a variable to receive the resulting text string

inputString text; the string to be queried

start integer; the position in the string where the substring starts

length integer; the number of characters to be extracted.

Effects

result is updated with length characters from inputString starting at character number start. Note that start counts from 0.

Example

```
...
let <var1> = "The quick brown fox";
let <var2> = substring <var1> 4 5;      // produces
"quick"
```

symbol

Function

Gets a symbol value from the PCE initialization file or start-up command line.

Syntax

```
let result = symbol "symbolName ";
```

Parameters

result a variable to receive the resulting text string

symbolName text; the name of a symbol in the PCE initialization file.

Effects

result is updated with the value of symbol symbolName. If a symbol is used that does not have a value, then an empty string is returned.

Comments

The symbol and its value must be defined either in the PCE initialization file or as part of the PCE start-up command line; see **Running PCE** on page 200 for details.

Example

```
...
//INI file contains <Symbol> runDate = "June Issue"
let <var1> = symbol "runDate";   //<var1> = "June Issue"
```

time

Function

Gets the current system time.

Syntax

```
let value = time;
```

Parameters

value. a variable to receive the time string.

Effects

value is updated with a string representing the system time when the command is issued. The format of this string is hh:mm:ss, for example: `13:06:56`.

Example

```
// Get and write date and time info to output
let <nowDate> = date format "ddmmmyyyy";
let <nowTime> = time;
let <outString> = "Date: %@nowDate Time: %@nowTime";
write 1 item into file 0 from <outString>;
```

TLE

Function

Retrieves an index value from an AFP page.

Syntax

```
let value = TLE name in afpPage ;
```

Parameters

value a variable to receive the value of the named index attribute

name text; the name of an AFP index attribute to be retrieved

afpPage a variable or array element containing a single AFP page.

Effects

afpPage is searched for an index attribute of name. If found, value is updated with the attribute value. If a match is not found value will be updated with a zero length string (note: you can check for this – see example).

Comments

This command searches for an AFPDS Tag Logical Element (TLE) structured field in the variable. If your publication generates an AFP journal a TLE will be placed in each page where an appropriate journal entry is included in a document design. The index attribute name is provided by the first object associated with the journal entry and the actual value is provided by the second object. Refer to the Designer User's Guide for more information about creating journals.

Note that this command is only valid when working with AFPDS files. Using it with other file formats will cause unpredictable results.

Example

```
...
read 1 item from file <AFPInput> into <AFPPage>;
let <Title> = "Customer_Name";
let <Value> = TLE <Title> in <AFPPage>;
let <OK> = Length <Value>;
if <OK>;
  write 1 item into file <ReportFile> from <Value>;
else;
  trace "Index not matched";
end if;
```

TLE add

Function

Adds an index element to an AFP page or document.

Syntax

```
TLE add at {page|document} of attrib name value value
[qualifier sequenceNum .levelNum] to afpPage ;
```

Parameters

name text; the attribute name to be assigned to the index element

value text; the value to be assigned to the index element

sequenceNum integer; sequence number to be assigned to the index element

levelNum integer; level number to be assigned to the index element

afpPage a variable containing one or more AFP pages or a PCE document group containing AFP pages.

Effects

An AFPDS Tag Logical Element (TLE) structured field is inserted into afpPage using name as its attribute name and value as its index value. If the qualifier keyword is coded sequenceNum is assigned as the index sequence number and levelNum as the level number.

If page is specified the TLE will be added within each page stored in afpPage. If document is specified the TLE will be added immediately before the first page stored in afpPage.

Comments

If you specify document you must ensure that all the pages making up the AFP document are stored in the variable when the TLE is added. When working at the document level you should normally read the pages into a PCE document group to ensure the header containing the TLE is not excluded.

The qualifier keyword expects two parameters separated by a decimal point. The value to the left of the decimal point is treated as the index sequence number and the value to the right becomes the level; e.g. 009.010 would produce a sequence number of 9 and a level of 10.

Example

```
...
//Read document pages into document group
read 10 pages from <InFile> into <Group>;
add document name of "Report_Group" to <Group>;
//Add a new document level TLE before these pages
TLE add at document of attrib "Doc_start" value
"First_doc" qualifier 123.456 to <Group>;
//Delete any page level TLEs with name ’Phase1’
TLE delete at page of attrib "Phase1" in <Group>;
//Add new page level TLEs
TLE replace at page of attrib "Phase2" value "First_pages"
 qualifier 123.456 to <Group>;
write 1 item into file <AFPOutput> from <Group>;
```

TLE delete

Function	Deletes an index element from an AFP page or document.
Syntax	`TLE delete at {page|document} of attrib name in afpPage;`
Parameters	name text; the attribute name of the index element to be deleted afpPage a variable containing one or more AFP pages or a PCE document group containing AFP pages.
Effects	AFPDS Tag Logical Element (TLE) structured fields in afpPage using name as the attribute name are deleted. If page is specified PCE will search for and delete matching fields in the AFP page headers. If document is specified PCE will search for and delete matching fields only from the document header.
Comments	If you specify document you must ensure that all the pages making up the AFP document are stored in the variable when the TLE is added. When working at the document level you should normally read the pages into a PCE document group to ensure the header containing the TLE is not excluded.
Example	See **TLE add** on page 161.

TLE replace

Function	Amends an index element in an AFP page or document.

Syntax	`TLE replace at {page|document} of attrib name value value`
	`[qualifier sequenceNum .levelNum] in afpPage ;`

Parameters	name text; the attribute name to be assigned to the index element value text; the value to be assigned to the index element sequenceNum integer; sequence number to be assigned to the index element levelNum integer; level number to be assigned to the index element afpPage a variable containing one or more AFP pages or a PCE document group containing AFP pages.
Effects	afpPage is searched for AFPDS Tag Logical Element (TLE) structured fields using name as the attribute name. If found the index value is replaced by value. If the qualifier keyword is coded sequenceNum and levelNum are used to replace the index sequence number and level number respectively. If page is specified the actual AFP pages are searched and, if appropriate, updated. If document only the document header is searched.
Comments	Body Text
Example	See **TLE add** on page 161

trace

Function

Writes a message to PCE trace output.

Syntax

```
trace message ;
```

Parameters

message text; the string to be output.

Effects

message is written to the PCE trace and log files assuming they have been assigned to the job.

Comments

The message text may contain variable names provided the correct syntax rules are followed. See **“Assigning values” on page 59** for details.

See **Running PCE** on page 200 for information about trace and log files for a PCE job.

Example

```
...
// Check the number of items read...
let <ReadResult> = <ReadNumber> lt 5;
// ... and write diagnostic message
if <ReadResult>;
  trace "There were less than 5 items read";
else;
  trace "There were @@<ReadNumber> items read";
end if;
```

translate

Function

Translates a text string using a Generate translation table.

Syntax

```
translate textToTrans using table tableId ;
```

Parameters

textToTrans a variable containing text; the string to be translated

tableId integer; the number assigned to the required table in the translation tables file.

Effects

The contents of textToTrans are overwritten with the results of translating each character code point of the original string to the appropriate character code point from the Generate translation table identified by tableId.

Comments

For details about translation tables files, refer to the “Generate Translation Tables” technical document on the **EngageOne Compose Support site**.

Example

```
...
translate <baseText> using table 22;
replace "*text to replace*" in <page> with <baseText>;
```

uservalue

Function

Gets a user defined value from the active <Preferences> section of the PCE initialization file.

Syntax

```
let value = uservalue keywordNum ;
```

Parameters

value. a variable to receive the preferences value

keywordNum integer; a reference to the required user value keyword.

Effects

value is updated with the value assigned to the keywordNumth 'UserValue' keyword in the currently active <Preferences> section of the PCE initialization file. For instance, if keywordNum = 3 then the value assigned to the UserValue3 keyword will be used.

Comments

User values are always integers. This function cannot return a string.

Example

```
...
// Set which INI file preferences section to use
set preferences to 3;
 // Get and write 7th user value in current section
let <prefValue> = uservalue 7;
write 1 item into file 0 from <prefValue>;
```

value

Function

Converts a number value held as a string to a number type.

Syntax

```
let result = value numberString ;
```

Parameters

result a variable to receive the resulting number

numberString text; a number value held as a string.

Effects

numberString is converted to a proper number type (if possible) and stored in result.

Comments

You may need to use this if you have first stored a numeric value in a variable using a text function.

Example

```
...
let <number> = substring "My number is 12" 14 2;  //number
 = "12"
let <realNumber> = value <number>;

let <intNumber> = value "99";
```

write

Function

Reads one or more items from a variable to a file.

Syntax

```
write count {items|pages} into file fileRef from var ;
```

Parameters

count integer; number of data items to write

fileRef integer; handle of a file opened for writing

var variable or array containing the data to write.

Effects

count items of data stored in var are written to the file referenced by fileRef.

If var is not an array the entire contents are written; count must always be 1 and any other value is ignored.

If var has been declared as an array count indicates the number of array elements to be written from the variable. You may specify the array element from which writing is to start. For example:

```
write 5 items into file 1 from <afpPages>(3);
```

will write the data from 5 array elements in <afpPages> starting with element 3. If you do not specify an element writing is assumed to start with element 0.

If an error occurs a return code from this function will be stored as system value <sys_last_error> which can be interrogated using the **on error call** on page 134 procedure.

Comments

Where multiple pages have been stored in a single array element count should be 1.

Do not use this command to write to a Document Interchange Journal (DIJ) file. Use **write DIJentry** on page 171.

Example

```
Declare <afpPages>(4);
...
// Rerun for summary page (1)
if <rerunType1>;
  write 1 page into file 1 from <afpData>(0);
end if;
// Rerun for details pages
if <rerunType2>;
  let <pageCount> = <baseCount> + 2;
  write <pageCount> pages into file 1 from <afpData>(1);
end if;
```

write DIJentry

Function

Writes a Document Interchange Journal entry to file.

Syntax

```
write DIJentry into file fileRef from dijRecord ;
```

Parameters

fileRef integer; handle of a file opened for writing

docGroup variable contain a DIJ entry.

Effects

```
dijRecord is written to fileRef.
```

If an error occurs a return code from this function will be stored as system value <sys_last_error> which can be interrogated using the **on error call** on page 134 procedure.

Comments

The file to be receive the DIJ entry must have been **open** on page 136ed as type DIJ. You may only write one entry at a time, i.e. dijRecord must contain a single DIJ entry.

Example

See **change DIJelement** on page 90.

Composition Edit Commands

This section details the composition edit (CE) commands that may be used with the begin ce/end ce construct within a PCE script file. CE commands allow you to add a limited set of new presentation elements to an existing composed page of output datastream. They have a different format from the main PCE script file statements that reflects the specialized nature of their function.

Positioning Concepts

All composition edit positions are defined as a meeting point between an X coordinate (‘across’ direction) and a Y coordinate (‘down’ direction) on a logical page. The 'top left' corner of the logical page is known as the logical page origin, i.e. X=0, Y=0, and is the point from which all positions are measured.

Logical pages are positioned within a physical page definition that reflects the media to be used on the actual output device. There may be more than one logical page on each physical page depending on how the Generate job that produced the output being manipulated was configured.

For most datastreams the physical page origin normally coincides with the origin of the first logical page (top left). For PostScript and PostScript variants however, the physical page origin is the bottom left corner. When manipulating PostScript or where the datastream contains multiple logical pages per physical page you will need to use the Set Physical Page Size (SPPS) command so that PCE can compensate coordinates accordingly. You must call this command before using commands that position objects on the page.

The current print position defines the position on the page where the next presentation element (text, line, etc.) will be placed. The coordinates used to specify the current position are always relative to the top left corner of the current logical page.

A Set Current Print Position (SCPP) command should be specified before each use of a composition command. If this is not done PCE will assume a print position of X=0, Y=0.

The unit of measure used for all CE commands is expressed in inches only. The maximum valid measurement for CE commands is 999.999 inches.

Resources

When adding new text or graphics to pages you need to be aware if your target output environment expects embedded or referenced font and image resources.

If resources are not embedded fonts and images are referenced by name only within the datastream and the required resource files (or suitable defaults) are assumed to be available on the target device.

For embedded resources you may only work with font and images that are already embedded in one of the output datastreams you are working with or which are available in the HIP files defined in the PCE initialization file.

Assuming you have chosen to embed resources in your job options, Generate will place them in the file header of the output datastream. If you are only working with a single datastream file as input to your PCE script this header will, by default, automatically be copied to any output files you create. In this scenario the existing resources are automatically available for use when creating new objects with CE commands.

If you need to use resources that are not embedded in your input files – i.e. you want to work resources from a HIP file – or you are intending to merge pages from different datastreams together you will need to specifically ensure that the file header of output files contains all the required resources. In these situations use the PCE get resources command to set the source to be used when gathering resources.

All output datastream files created by a PCE script will always share the same list of resources.

You can refer to a resource by the any of the names by which it is known in the existing output datastreams being manipulated and the HIP files referenced in the PCE initialization file. In most cases this will simply be the base name of the font or image, for instance X0T05500 or PIC001. However, when referring to fonts for PostScript or PostScript variants the font syntax can be more complex: For example:

```
Arial-bold.10
Courier.12
TimesNewRoman-oblique.9
```

For this reason you should first code the PCE **font** command where possible to establish the valid reference names. See **“font” on page 94** for details.

Using Color

By default, all new elements created by CE commands will be presented in black. If you want use other colors you must code one or more Set Color commands (COLR) in the appropriate sequence within the script.

You may specify colors either using keywords for one of 8 'standard' colors or by specifying a RGB value for all other colors.

Be aware that your intended output device may have limited or no color support. Specifying unsupported colors may have unpredictable results. Note that for some datastreams colors defined in COLR commands may be overriding or ignored by settings in the PCE initialization file.

Order of Commands

Composition edit commands must conform to the following order:

- If you intend to call image names (via the PI command) or overlay names (via PPO) you must include a Define Image List command (DIL) and/or a Define Page Overlay List command (DPOL) before the appropriate include commands (PI and/or PPO).
- If you are manipulating PostScript output or a datastream with multiple logical pages per physical page you must code a Set Physical Page Size command (SPPS) before any SCPP or other commands containing coordinates.

- If you need to create presentation elements using colors other than black you must use a Set Color command (COLR) before the relevant presentation commands. The color specified in this command stays in force until the next COLR is encountered.
- A Set Current Print Position command (SCPP) must precede any command that places an object other than PI or PPO;
- If an STL command is to follow the SCPP use a Set Text Presentation command (STP) to set the required orientation.

  Syntax

All composition edit commands have a fixed structure and are position sensitive. All commands are introduced with an equals sign (=) in column 1 and are terminated with a double semicolon (;;). Continuation records may be used if a particular command is longer than 80 bytes.

Commands are identified by mnemonic keywords in positions 2 - 5. All keywords must appear in upper case. Both keywords and parameters must occupy a fixed number of positions and trailing spaces must be used to pad the blank positions where necessary.

You can use a PCE variable as part or all of a string in any CE command. Such variables must be prefixed with the characters %@. Where necessary, such variables must include space padding to allow for position sensitive elements

The measurement values for positioning commands are always in inches and are expressed as a signed number with three decimal places, i.e. a format of ±`nnn.nnn.` This is known as a CE measurement. All numeric positions must be specified in full, e.g. '000.010'. The sign can be omitted if the intended value is positive but where this is the case the position must be padded with a space.

The following examples illustrate these points. Required spaces are indicated by •:

```
=SCPP••001.500••002.000;;
=DHR••+003.000••000.050;;
=STP••90;;
=STL••X0T05501••var1 is %@<var1>;;
=PI•••S1LOGO••••001.000••001.000;;
```

COLR – Set Color

Function

Sets the color to be used for drawing subsequent presentation elements.

Syntax

```
=COLR•{doc1color|Rnnnnnn};;
```

Parameters

doc1Color integer; indicates one of the standard Generate colors. Choose from:

1 - Black

2 - Blue

3 - Brown

4 - Green

5 - Pink

6 - Red

7 - Cyan

8 - Yellow

`R`nnnnnn RGB color; nnnnnn indicates the red, green and blue values specified as hex numbers:

Effects

Presentation elements generated by commands following COLR will be created using the color indicated (until a further COLR is encountered).

Example

```
=COLR  1;;
=SCPP  000.500  001.500;;
=DHR  +000.250 000.020;;
=COLR  R0f0f0f;;
=SCPP  000.500  003.500;;
=DHR  +000.250 000.020;;
```

DBX – Draw Box

Function

Draws a box at the current print position, optionally shaded.

Syntax

```
=DBX••shade •thickness •width •height ;;
```

Parameters

shade three digits representing a percentage (i.e. 001...100); the percentage of shading required within the box

thickness CE measurement; the thickness of the sides of the box. If this value is set to zero no sides are drawn.

width CE measurement; the width (X direction) of the box

height CE measurement; the height (Y direction) of the box.

Effects

A command is inserted into the current page to draw a box of width x height dimensions and a fill percentage of shade. Its top left corner is determined by the last SCPP command to be processed. The box sides (if any) are drawn with a solid rule with the thickness specified. Both box sides and shading will use the color specified in the last COLR command to be processed or black if no such command has been encountered.

Comments

The box is drawn so that the lines are within the dimensions specified. For example, if a box is 1" in height and 0.5" in width with a rule thickness of 0.2", the total height of the box would be 1" and the width would be 0.5".

Example

```
=SCPP  001.000  001.000;;
=DBX  005 000.020 002.000 001.000;;
```

DHR – Draw Horizontal Rule

Function

Draws a horizontal line at the current print position.

Syntax

```
=DHR••length •thickness ;;
```

Parameters

length CE measurement; the length of the rule

thickness CE measurement; the line thickness.

Effects

A command is inserted into the current page to draw a solid horizontal rule starting at the position specified by the last SCPP command to be processed. If length is a positive value the rule is drawn from left to right. If negative it is drawn from right to left.

Comments

The thickness of the rule is always drawn down the page.

Example

```
=SCPP  001.000  001.000;;
=DHR  +003.500 000.050;;
```

DIL – Define Image List

Function

Declares the names of image resources to be used with the current Begin CE/End CE construct.

Syntax

```
=DIL••image001 •image002 •image003...;;
```

Parameters

image... text; the name by which an image resource is known to Generate. This should normally be the name used for the resource in the Designer. If the name is specified as a quoted string it may be any length. If not it must be 8 characters padded with spaces if necessary.

Effects

The image names specified are made available for use with subsequent PI commands.

Comments

Up to 127 image names may be specified, each separated by a space. Only images already present in the HIP files being used with the PCE job may be referenced. The resource names must match the references used in the HIP exactly.

Example

```
=DIL  S1LOGOA1 S1LOGO   "Marketing July" %@<Img1>;;
=PI   S1LOGO1  001.500  001.500;;
```

DO – Do composition function (barcodes)

Function	Performs a predefined composition function using the input provided and outputs the result at the current print position. In the current version of PCE this is limited to the formatting of barcode output.

Syntax

```
=DO•••BARCODE•TYPE {•OPTION =PARM •OPTION
=PARM...}•USING•VALUE ;;
```

Parameters

TYPE is one of: `POSTNET | PLANETCODE | 2OF5 | 3OF9 | 3OF9CHECKSUM | CODE128A | CODE128B | CODE128C | PDF417 | DataMatrix | MaxiCode | IntMail | InfoMail:`

OPTION=PARM You may need to specify one or more settings depending on the type of barcode being used. Use the tables below for reference.

VALUE the input to the function. You may include a value stored in a PCE variable by using the format %@<VARNAME>.

Options for PostNet, PlanetCode

Option	Parameters	Default
`Direction`	`HORIZONTAL | VERTICAL`	`HORIZONTAL`

Options for 2of5, 3of9, 3of9Checksum, Code128n (continued)

Option	Parameters	Default
`Direction`	`HORIZONTAL | VERTICAL`	`HORIZONTAL`
`Density`	`HIGH | MEDIUM | LOW`	`MEDIUM`

Options for PDF417

Option	Parameters	Default

Options for PDF417

Option	Parameters	Default
Direction	HORIZONTAL | VERTICAL	HORIZONTAL
Font Aspect	1:2 | 1:3 | 1:4 | 1:5	1:2
Security	0-8	0
Barcode Aspect	0-100	0
Start Mode	ALPHA | LOWER | MIXED | PUNCTUATION | NUMBER	ALPHA

Options for DataMatrix

Option	Parameters	Default
Direction	HORIZONTAL | VERTICAL	HORIZONTAL
Mode	SQUARE | FLAT	SQUARE
Security	IF MODE = SQUARE 0-23 IF MODE = FLAT 0-6	0
	Size in points	varies – one used in input

Options for DataMatrix

Option	Parameters	Default
`Fontsize`		

Options for MaxiCode

Option	Parameters	Default
`Mode`	`UPS, 2-6`	none
`Border Width`	`0-999.999` (inches)	`0`
`Background color`	`BLACK | BLUE | BROWN | GREEN | PINK | RED | CYAN | YELLOW | DARK BLUE | DARK GREEN | TEAL | GRAY | MUSTARD | ORANGE | PURPLE | WHITE Rrrggbb` – RGB value with parameters as hex codes	none
`Postal Code`	`string`	none
`Country Code`	`string`	none
`Service Class`	`string`	none
`Tracking Number`	string	none
	Valid for Modes 2-6 only. string	none

Options for MaxiCode

Option	Parameters	Default
`Origin Carrier`		
`Shipper ID`	Valid for UPS Mode only. string	none
`Pickup Date`	Valid for UPS Mode only. string	none
`Shipment ID`	Valid for UPS Mode only. string	none
`Package Number`	Valid for UPS Mode only. string	none
`Package Count`	Valid for UPS Mode only. string	none
`Weight`	Valid for UPS Mode only. string	none
`Street`	Valid for UPS Mode only. string	none
`City`	Valid for UPS Mode only. string	none
`State`	Valid for UPS Mode only. string	none

Options for MaxiCode

Option	Parameters	Default
`Address Validation`	`YES | NO`	`YES`

Effects

A barcode of the requested type is generated using the assigned value. A command is inserted into the current page to add the barcode using the position specified by the last SCPP command to be processed as the 'top left' corner.

Example

```
=SCPP  001.000  001.000;;
=DO   BARCODE POSTNET DIRECTION=VERTICAL USING
"1234567890";;
=SCPP  005.100  001.000;;
=DO   BARCODE DATAMATRIX MODE=FLAT SECURITY=6 USING
%@<BCDATA>;;
```

DPOL – Define Overlay List

Function

Declares the names of overlay resources to be used with the current Begin CE/End CE construct.

Syntax

```
=DPOL•overlay1 •overlay2 •overlay3...;;
```

Parameters

overlay... text; the name by which an overlay resource is known to Generate. This should normally be the name used for the resource in the Designer. If the name is specified as a quoted string it may be any length. If not it must be 8 characters padded with spaces if necessary.

Effects

The overlay names specified are made available for use with subsequent PPO commands.

Comments

Up to 127 overlay names may be specified, each separated by a space. Only overlays already present in the HIP files being used with the PCE job may be referenced.

Example

```
=DPOL O1OVER   O1OVERXX "Response Form" %@<Over>;;
=PPO  O1OVERXX  001.500  001.500;;
```

DVR – Draw Vertical Rule

Function

Draws a vertical rule at the current print position.

Syntax

```
=DVR••length •thickness ;;
```

Parameters

length CE measurement; the length of the rule

thickness CE measurement; the line thickness.

Effects

A command is inserted into the current page to draw a solid vertical rule starting at the position specified by the last SCPP command to be processed. If length is positive the rule will be drawn downwards (i.e. from top to bottom) from the Current Print Position. If negative it will be drawn upwards (i.e. from bottom to top).

Comments

The thickness of the rule always goes across the page from left to right.

Example

```
=SCPP  001.000  001.000;;
=DVR  +003.500 000.050;;
```

NOP - No Operation

Function

Adds a NOP Comment instruction to an AFP Page. The NOP is inserted before the Active Environment Group.

Syntax

```
=NOP••content;;
```

Parameters

content text; the string to be added. This can contain references to PCE variables using the %@ introducer if required

Effects

A NOP instruction is inserted into the current AFP page with the specified content.

Comments

Example

```
=NOP
My_First_Test_String%@<MyVariable>Second_String%@<MyVariable>;;
```

OUN - Ouput User Note

Function

Adds a NOP Comment instruction to an AFP Page. The NOP is inserted inside the Presentation Text object.

Syntax

```
=OUN••content;;
```

Parameters

content text; the string to be added. This can contain references to PCE variables using the %@ introducer if required

Effects

A NOP instruction is inserted into the current AFP page with the specified content.

Comments

If no Presentation Text object is created from the CE command list, this instruction has no effect. If you do not require the NOP to be inside Presentation Text, use the NOP command.

Example

```
 =OUN
My_First_Test_String%@<MyVariable>Second_String%@<MyVariable>;;
```

PBIM – Place Barcode – Intelligent Mail

Function

Inserts an Intelligent Mail barcode.

Syntax

```
=PBIM••orientation •fullheight •trackerheight •barwidth
 •density•string ;;
```

Parameters

orientation rotation of the barcode: 0 – 0 degrees (left to right) 1 – 90 degrees (top to bottom) 2 – 180 degrees (right to left, upside-down) 3 – 270 degrees (bottom to top)

fullheight height of the full bar in inches or fractions of an inch

trackerheight height of the tracker bar in inches or fractions of an inch

barwidth width of the bar in inches or fractions of an inch

density width of the space between the bars in inches or fractions of an inch

string string to be encoded into the barcode. Can be enclosed in double quotes.

Effects

A command is inserted into the current page to draw an Intelligent Mail barcode starting at the position specified by the last SCPP command to be processed.

Comments

The barcode is drawn using rectangles, so no font is required.

Example

```
=SCPP  005.100  001.000;;
=PBIM  0 000.125 000.039 000.015 000.012 01234567094987654321;;
```

PI – Place Image

Function

Inserts an image resource into a page.

Syntax

```
=PI•••image [•xOffset •yOffset];;
```

Parameters

image text; the name by which an image resource is identified in the HIP files assigned to the current PCE job and which has previously been declared in a DIL command. If the name is specified as a quoted string it may be any length. If not it must be 8 characters padded with spaces if necessary

...Offset CE measurements. The coordinates for positioning the top left corner of the image.

Effects

A command to include image on the current page at xOffset/yOffset (or at the current print position if these parameters are omitted) is inserted.

Comments

A command to include image on the current page at xOffset/yOffset (or at the current print position if these parameters are omitted) is inserted.

Example

```
=PI   S1IMAG01  001.500  001.500;;
=PI   %@<Img1> +003.000 +002.000;;
```

PPO – Place Page Overlay

Function

Inserts an overlay resource into a page.

Syntax

```
=PPO••overlay [•xOffset •yOffset];;
```

Parameters

overlay text; the name by which an overlay resource is identified in the HIP files assigned to the current PCE job and which has previously been declared in a DPOL command. If the name is specified as a quoted string it may be any length. If not it must be 8 characters padded with spaces if necessary

...Offset CE measurements. The coordinates for positioning the top left corner of the overlay.

Effects

A command to include overlay on the current page at xOffset/yOffset (or at the current print position if these parameters are omitted) is inserted.

Example

```
=PPO  O1OVER01  001.500  001.500;;
=PPO  %@<Over> +003.000 +002.000;;
```

SBT – Set Boxed Text

Function

Creates a 'white' text element centered in a box.

Syntax

```
=SBT••font •width •height •content ;;
```

Parameters

font text; the name by which a font resource is identified within the HIP files assigned to the current PCE job. If the name is specified as a quoted string it may be any length. If not it must be 8 characters padded with spaces if necessary

width CE measurement; the width (X direction) of the box

height CE measurement; the height (Y direction) of the box

content text; the string to be included in the box. This can contain references to PCE variables using the %@ introducer if required.

Effects

A command is inserted into the current page to draw a box of width x height dimensions with solid fill. Its top left corner is determined by the last SCPP command to be processed. The fill color will be the color specified in the last COLR command to be processed or black if no such command has been encountered.

Content will be presented using the selected font and centered within the box. The text has no color and will show as the current paper color unless it overlays other elements. No wrapping of text occurs: if the font height is greater than the box height, or if the text width is greater than the box width, the text will extend beyond the boundaries of the box.

Comments

For PostScript pages any font can automatically be reversed and therefore you may specify any font known to the HIP file.

AFP fonts cannot be reversed automatically and customized reversed fonts are normally created where required. In such fonts each character raster must extend to the maximum ascender and descender of the font otherwise this function will not work properly; a symptom of this would be white bars appearing between characters. Refer to your product supplier for more information regarding the use of reversed fonts for AFP.

Example

```
=SCPP  001.000  001.000;;
=SBT  "Arial-bold.10" 005.000 001.000 "Amount to pay:
%@total";;
```

SBTR – Set Boxed Text Right Justified

Function

Creates a right justified 'white' text element centered in a box.

Syntax

```
=SBTR•font •width •height •margin •content ;;
```

Parameters

font text; the name by which a font resource is identified within the HIP files assigned to the current PCE job. If the name is specified as a quoted string it may be any length. If not it must be 8 characters padded with spaces if necessary

width CE measurement; the width (X direction) of the box

height CE measurement; the height (Y direction) of the box

margin CE measurement; the amount content is offset from the right side of the box

content text; the string to be included in the box. This can contain references to PCE variables using the %@ introducer if required.

Effects

A command is inserted into the current page to draw a box of width x height dimensions with solid fill. Its top left corner is determined by the last SCPP command to be processed. The fill color will be the color specified in the last COLR command to be processed or black if no such command has been encountered.

Content will presented using the selected font and will be right justified within the box offset by margin. The text has no color and will show as the current paper color unless it overlays other elements. No wrapping of text occurs: if the font height is greater than the box height, or if the text width is greater than the box width, the text will extend beyond the boundaries of the box.

Comments

For PostScript pages any font can automatically be reversed and therefore you may specify any font known to the HIP file.

AFP fonts cannot be reversed automatically and customized reversed fonts are normally created where required. In such fonts each character raster must extend to the maximum ascender and descender of the font otherwise this function will not work properly; a symptom of this would be white bars appearing between characters. Refer to your product supplier for more information regarding the use of reversed fonts for AFP.

Example

```
=SCPP  001.000  001.000;;
=SBTR "Arial-bold" 005.000 001.000 "Amount to pay:
%@total";;
```

SCPP – Set Current Print Position

Function

Sets the position to be used when inserting subsequent presentation elements.

Syntax

```
=SCPP•[+|-|•]xxx.xxx •[+|-|•]yyy.yyy ;;
```

Parameters

xxx.xxx, yyy.yyy CE measurements. The position coordinates in relation to the top left corner of the logical page. Must be preceded by a sign or an extra space.

Effects

Subsequent commands that create presentation elements will be positioned using the coordinates specified in this command.

Comments

Negative values will result in a position outside the logical page boundary and should therefore not be specified. An absolute position is used when no sign indicators precede the coordinates. A relative position is used for any coordinate that contains a sign indicator. Note that when relative coordinates are specified it is the user's responsibility to ensure that the position is set within the logical page boundary.

Example

Example of setting an absolute print position:

```
=SCPP  001.000  004.000;;
```

Example of setting an absolute X print position with a relative Y print position:

```
=SCPP  001.000 +002.000;;
```

Example of setting a relative X print position with an absolute Y print position:

```
=SCPP +001.500  003.000;;
```

Example of setting a relative print position for both coordinates:

```
=SCPP +001.500 -002.000;;
```

SPPS – Set Physical Page Size

Function

Specifically defines the size of physical pages.

Syntax

```
=SPPS••{width •height |name };;
```

Parameters

width CE measurement; the width (X direction) of the page

height CE measurement; the height (Y direction) of the page

name text; a keyword describing the page size. Valid options are: A4, B4, B5, USLETTER, USLEGAL.

Effects

PCE compensates positioning commands using the size information given.

Comments

You will need to code this command when you are processing PostScript and PostScript variants or where the datastream being manipulated contains multiple logical pages per physical page.

Where used, this command must be placed before any positioning or drawing commands, including SCPP. If this command is not coded the default page size is A4.

Note that this command is only used when calculating the offset of new presentation elements; it does not affect logical page sizes in any way.

Example

```
=SPPS +008.500 +011.000;;
=SPPS  A4;;
```

STL – Set Text Line

Function

Adds a text string using the specified font at the current print position.

Syntax

```
=STL••font •content ;;
```

Parameters

font text; the name by which a font resource is identified within the HIP files assigned to the current PCE job. If the name is specified as a quoted string it may be any length. If not it must be 8 characters padded with spaces if necessary

content text; the string to be added. This can contain references to PCE variables using the %@ introducer if required.

Effects

A command is inserted into the current page that will present content at the current print position in the font specified. The text is always presented as a single line; i.e. no wrapping occurs.

The text will be presented at 0° relative to the logical page unless an STP command with a specifying a different orientation has been processed.

Comments

If the string is too long for the available logical page space the subsequent behavior will depend on the output device used.

Example

```
=SCPP  001.000  001.000;;
=STP 270;;
=STL  X0T05500 "Detach here";;
```

STP – Set Text Presentation

Function

Sets the orientation to be used with subsequent STL commands.

Syntax

```
=STP••[0|90|270];;
```

Effects

Text added using subsequent STL commands will be presented in the orientation coded.

Comments

Presentation commands other than STL are not affected. An orientation of 180° is not supported.

Example

```
=SCPP  001.000  001.000;;
=STP 90;
=STL  X0T05500 "Page 1 of 2";;
```

Script file sample

This example of a PCE script file is intended to post-process a single AFPDS 240 datastream file. The production system is assumed to be Windows.

The objectives of the application is to add a new box and text to the pages of an existing datastream and split the output into two new files based on whether or not the page count is odd.

```
// Declare the variables needed in the program

DECLARE <Statement>;     // Statement data is stored here
DECLARE <PageCnt>;       // Number of pages
DECLARE <N>;             // Pages (to be) read
DECLARE <Done>;          // TRUE at end of file
DECLARE <Odd>;           // Variable to split odd and even numbers
DECLARE <App>;           // Variable to hold name of Publication
DECLARE <Infile>;        // Variable to hold name of Input Publication
DECLARE <Outfile1>;      // Variable to hold name of 1st Output
Publication
DECLARE <Outfile2>;      // Variable to hold name of 2nd Output
Publication

// Declare the procedures
DECLARE PROCEDURE <Main> IS MAIN;

BEGIN PROCEDURE <Main>;

    // Resolve Input Filename
    LET <APP> = SYMBOL "APPNAME";
    LET <INFILE> = <APP> + "240.afp";

    //Resolve Output Filenames
    LET <oUTFILE1> = <APP> + "oddpage2.afp";
    LET <oUTFILE2> = <APP> + "evenpage2.afp";

    // Open the input file
    OPEN <INFILE>  FOR INPUT AS FILE 1 WSAFP/AFPDS;

    // Open output files
    OPEN <OUTFILE1> FOR OUTPUT AS FILE 1 WSAFP/AFPDS;
    OPEN <OUTFILE2> FOR OUTPUT AS FILE 2 WSAFP/AFPDS;

    // Initialize variables
    LET <PageCnt> = 0;
    LET <Odd> = 0;

    // Loop for each customer...
    BEGIN LOOP;
```

```
        // Establish if odd or even page
        LET <Odd> = <Odd> NE 1;

        // Quit when 21 pages processed
        LET <Done> = <PageCnt> EQ 21;
        EXIT LOOP WHEN <Done>;

        // Read the composed print data
        LET <N> = 1;

        // Number of pages read is returned in <N> below
        READ <N> PAGES FROM FILE 1 INTO <Statement>;
        LET <Done> = <N> LT 1;

        // Exit if page not read
        EXIT LOOP WHEN <Done>;

        // or update page count if successful
        LET <PageCnt> = <PageCnt> + 1;

         // Add new elements to the page via composition edit commands
         // The CE commands are position sensitive
         BEGIN CE INTO <Statement>;
            // Set print position
           =SCPP  000.500  005.700;;
            // Draw box
           =DBX  000 000.100 007.000 005.500;;
           // Set print position
           =SCPP  000.600  006.000;;
           // Set text presentation
           =STP  0;;
           // Add text line
           =STL  02 This box was added by PCE;;
         END CE;

        // Write to the appropriate output files
        IF <Odd>;
            TRACE "Copying page @@<PageCnt> statement to file 1";
            WRITE 1 ITEM INTO FILE 1 FROM <Statement>;
        ELSE;
            TRACE "Copying page @@<PageCnt> statement to file 2";
            WRITE 1 ITEM INTO FILE 2 FROM <Statement>;
        END IF;

    END LOOP;

END PROCEDURE;
```

6 - Running PCE

A PCE job is controlled by its initialization file which contains resources definitions and other control information used by the job.

In this section

PCE resources

A PCE job uses the following resources:

Script file

Mandatory – this is the program code that describes what PCE is to do when a job is run. Refer to **Programming PCE** on page 62 for details.

Initialization file

Mandatory – this provides information about the PCE environment including regional and system settings.

Output datastream(s)

The file(s) to be manipulated by PCE. Files that are input to PCE must have been produced by Generate. During a typical PCE process, pages from the datastreams are read to memory, manipulated as required and then output to a new file. Both input and output files are identified using an open command in the PCE script itself.

Journal files

Many publications create one or more journal files to act as an index into the pages created in the output datastream. PCE can use these to locate and extract specific pages. It can also write new or updated journal information if required. Journal files are identified using an open command in the PCE script itself.

HIP file

This is the file containing the instructions and resources for a publication as created by the Designer. If you intend to add new presentation objects to pages you will need to identify the HIP file that was used when the output datastream was initially created by Generate. This is specified in the <Files> section of the initialization file.

Translation tables file

Mandatory – this contains tables that are used when PCE converts text data. A standard translation tables file is provided with product distribution material. Custom files may be supplied in some circumstances. The file to be used is referenced in the <Files> section of the initialization file.

Messages file

Mandatory – this contains the text of the diagnostic messages that may be issued by PCE and is supplied with product distribution material. It must be referenced in the <Files> section of the initialization file.

User exit control file

If you need to call functions from external programs as part of the PCE process you will need create this file to identify the modules required and specify the type of each function to be called. The control file itself is referenced in the <Files> section of the initialization file.

Text substitution file

If you want to use the PCE mapp command this file provides the text strings to be returned according to the parameter used. It is referenced in the <Files> section of the initialization file. See **mapp** on page 122 for more information.

Exception dictionary

If you want to use the PCE mixc command this file includes the correct casing for non-standard text strings. It is referenced in the <Files> section of the initialization file. See **mixc** on page 125 for more information.

Creating an initialization file

A PCE initialization file (INI) is a text file that can be created using any standard text editor.

The file format consists of several distinct sections in which can be coded a range of keywords and their associated parameters. Section names are enclosed in angle brackets. All code following a section name is considered to belong to that section until the next section name is encountered. For example:

```
<Files>
Messages=messages.dat
Input=gen\pce\scripts\job1.txt
TranslationTable=c:\gen\doc1ttab.ett
TextSubs=c:\gen\resource\myets.ets
HIP=c:\dochost\billjob.hip

<System>
AsciiToAfpds=27
;other System values in this include file
#include doc1static.ini

<Afpds>
DisableColor=DOWNGRADE

<Preferences1>
Day1=Dimanche
```

Sections and keywords within sections can be coded in any order. Most keywords have default parameters that are used if the keyword is not coded in the file.

Parameters are normally coded as literal values but can be specified dynamically if required by defining them wholly or partly as symbols. Values can be assigned to such symbols when starting the DOC1PCE program. Within the INI file symbols are referenced by coding the name to be used within percentage signs as in the example above. Symbol names can also be referenced within the PCE script itself by using the symbol command.

You can also use an #include statement to add the contents of an existing INI file to another. This allows you to reuse common INI settings. The contents of an include file are added at the point where the #include statement is encountered.

Note that If sections or keywords are repeated within an INI file or #include files the last element to be encountered will be used (working top-to-bottom).

Text following a semi-colon up to the start of the next line is as a comment.

At minimum a PCE INI must contain:

- a <Files> section with at least the Messages and Input keywords

- a <PrintDevice> section with at least the **PrintStream** and **Resolution** keywords

If your script uses functions that return regional information (such as date, day or monthname) you may often want to create one or more <Preferences> sections to indicate how the values are to be formatted. You may include up to 10 different sections: <Preferences0>, <Preferences1> … <Preferences9>.

If your script adds presentation objects to existing pages the INI will also often contain a section that allows you to customize some aspects of the output datastream being manipulated (for instance <AFPDS> or <PostScript>) and a <System> section for non-standard environment settings.

INI section summary

The following pages provide a reference for all available INI sections and their associated keywords.

<AFPDS>

This section contains customizable settings related to AFP datastreams. They are used only when PCE is adding presentation objects to existing pages.

Syntax and defaults:

```
<AFPDS>
DisableColor={YES|NO|DOWNGRADE} ;default NO
MaxPtxRecordSize=Number ;default 8200
MaxRuleThickness=Number ;default 32
BuildDefaultFormdef={YES|NO} ;default YES
UseExtendedMediumMap={YES|NO} ;default YES
```

Keywords and parameters:

DisableColor	By default, Generate includes color commands in AFP that conform to the specification for a full color AFP environment. If you are using an older AFP environment you may need specify either: YES – do not include any color commands; DOWNGRADE – use AFP commands suitable for a 16 standard color environment.
MaxPtxRecordSize	Number is an integer and indicates the maximum size of PTX structured fields used when adding new text to a datastream.
MaxRuleThickness	Number is an integer and indicates the maximum number of PELs to be used in individual commands when drawing new lines and boxes.
BuildDefaultFormdef	The default option **Yes** includes the form definition, F1G1DFLT, in resources that are built from hip files listed in the <Files> section. The form definition is not included if the option is set to **No**.
UseExtendedMediumMap	The default option **Yes** includes input tray and output bin settings with medium maps when they are embedded into the AFP datastream. If **No** is used the input tray and output bin settings are ignored.

Example:

```
<AFPDS>
DisableColor=YES
MaxPtxRecordSize=32000
MaxRuleThickness=64
BuildDefaultFormdef=YES
UseExtendedMediumMap=YES
```

<Exception>

This section defines the level of exception messages to be issued by PCE.

Syntax and defaults:

```
<Exception>
SuppressWarnMsg={YES|NO} ;default NO
SuppressInfoMsg={YES|NO} ;default NO
```

Keywords and parameters:

SuppressWarnMsg	If YES PCE will not issue any messages classed as warnings.
SuppressInfoMsg	If YES PCE will not issue any messages classed as information.

Example:

<Exception> SuppressWarnMsg=YES SuppressInfoMsg=YES

<Files>

This section identifies the standard PCE files to be used. Note that output datastream files to be manipulated by the job and their associated journal files are specified using an open command in the PCE script itself. File reference should be coded in the format required on the system that will run PCE; for example:

Windows

```
Input=c:\gen\pce\scripts\job1.txt
```

UNIX

```
Input=/gen/pce/scripts/job1.txt
```

z/OS

```
Input="GEN.PCE.SCRIPTS(JOB1)"
Input=DD:PCESCRPT
```

Syntax and defaults:

```
<Files>
Input=File ;mandatory, no default
Messages=File ;mandatory, no default
TranslationTable=File ;mandatory, no default
TextSubs=File ;optional
HIP=File ;optional
UserExit=File ;optional
ExceptionDictionary=File ;optional
TraceInfo=File ;optional
LogInfo=File ;optional
DOC1ecp=File ;optional
```

Keywords and parameters:

Input	File is the PCE script to be used by the job. This keyword must always be specified.
Messages	File is the PCE diagnostic messages file as provided with product distribution material. If possible code this keyword as early as possible in the INI file. Always ensure the correct version of the file is used. This keyword must always be specified.

TranslationTable	File is a PCE translation tables file. Typically this will be the generic file provided with product distribution material although custom files may be provided in some circumstances. This keyword must always be specified.
TextSubs	File is a Generate lookup table file (also known as a text substitution file). This is required if the PCE script includes the mapp command. See the mapp section of the **PCE command reference** on page 78 for more information including how to create a text substitution file.
HIP	File is a HIP file as created as part of a Publish task on the Designer. The HIP file(s) specified in the INI will need to contain all font and image resources that are referenced by the PCE script. You may code this keyword multiple times or use wild cards in the parameter to indicate multiple files.
UserExit	File is a user exit control file that provides linkages to external functions. This file is required if the PCE script includes the call userexit command – See the call userexit section of the **PCE command reference** on page 78 for details. For complete information about the user exit environment including how to create a user exit control file see **User exits** on page 272.
ExceptionDictionary	File is a Generate exception dictionary file used to provide proper casing of acronyms and other non-standard strings. This is required if the PCE script includes the mixc command. See the mixc section of the **PCE command reference** on page 78 for more information including how to create an exception dictionary file.
TraceInfo/LogInfo	If specified File will receive messages issued by the current PCE job in both cases. The trace file is always overwritten. The log file is appended.
DOC1ecp	File is the location of the DOC1ecp file containing the DBCS code pages required to support a PCE job on a non-ansi platform, such as in a non-Western production environment.

Examples:

Under z/OS:

```
<Files>
Messages=DD:MESSAGES
Input=DD:PCESCRPT
TranslationTable='USER001.TEMP(DOC1TTAB)'
```

Under Windows:

```
<Files>
Messages=messages.txt
Input=c:\gen\pce\scripts\job1.txt
HIP=gen\resources\*.hip
```

<Postscript>

This section contains customizable settings related to PostScript.

Syntax and defaults:

```
<Postscript>
Symbols="Controls " ;default "{}[]!~^$|"
RecordLength=Integer ;default 255
UseFormsForImages={YES|NO} ;default NO
```

Keywords and parameters:

Symbols

Controls indicate the nine symbols that are used as part of the syntax within the PostScript being manipulated. Specifically these are left brace, right brace, left square bracket, right square bracket, exclamation mark, tilde, circumflex accent, dollar sign and vertical line. If this keyword is coded you must specify the symbols as a text string or hex codes indicating the relevant code points. The order of symbols must always conform to that given and the parameter must always be enclosed in quotes. An example of the required hex format is as follows:

```
"x'7B',x'7D',x'5B',x'5D',x'21',x'7E',x'5E',x'24',x'7C'"
```

You must use the same parameter that was used when Generate originally created the PostScript being manipulated – i.e. the default or the Control symbols setting specified in the output device used when publishing a job.

RecordLength

Integer is the maximum length of any new PostScript records created by the PCE script. The value should be in the range 64–255 inclusive.

UseFormsForImages

When set to Yes then images will be placed in the Postscript formspace. You must use this when the option to cache images has been set in the Designer. The default is No.

Example:

```
<Postscript>
Symbols="()<>!~^$|"
RecordLength=200
UseFormsForImages=YES
```

<Preferencesx>

These sections allow you to define the format of date components and other regional settings which can be referenced by the PCE script. You can specify up to Preferences sections (0-9) which can be activated within the script. <Preferences0> is always active when a PCE job starts. Default values for this section are configured for international English.

Syntax and defaults:

```
<Preferences0> ;0 is the primary definition
Ordinal1=String ;default 1st
... ;other defaults 2nd,3rd,4th,etc.
Ordinal31=String ;default 31st
MonthName1=String ;default January
... ;other defaults February,March,etc.
MonthName12=String ;default December
MonthAbbrev1=String ;default Jan
... ;other defaults Feb,Mar, etc..
MonthAbbrev12=String ;default Dec
Day1=String ;default Sunday
... ;other defaults Monday,Tuesday, etc.
Day7=String ;default Saturday
NumericConvCode=String ;default ".,+-_"
NumericPadding=Char ;default ‘,’
UserValue1=Number ;default 0
...
UserValue16=Number ;default 0

<Preferences1> ;first alternate definition
...
<Preferences9> ;ninth alternate definition
...
```

Keywords and parameters:

Ordinal1…31

String is the text to return when an ordinal command is processed in the PCE script. The value of Ordinal1 will be returned where the script parameter is 1, Ordinal2 when the parameter is 2 and so on. The default values for Ordinal1-Ordinal31 in sequence are: 1st, 2nd, 3rd, 4th, 5th, 6th, 7th, 8th, 9th, 10th, 11th, 12th, 13th, 14th, 15th, 16th, 17th, 18th, 19th, 20th, 21st, 22nd, 23rd, 24th, 25th, 26th, 27th, 28th, 29th, 30th, 31st.

MonthName1…12

String is the text to return when a monthname command is processed in the PCE script. The value of MonthName will be returned where the script parameter is 1, MonthName when the parameter is 2 and so on. The default values for MonthName1-MonthName12 in sequence are: January, February, March, April, May, June, July, August, September, October, November, December.

MonthAbbrev1…12	String is the text to return when a monthabbrev command is processed in the PCE script. The value of MonthAbbrev1 will be returned where the script parameter is 1, MonthAbbrev2 when the parameter is 2 and so on. The default values for MonthAbbrev1-MonthAbbrev12 in sequence are: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec.
Day1…7	String is the text to return when a day command is processed in the PCE script. The value of Day1 will be returned where the script parameter is 1, Day2 when the parameter is 2 and so on. The defaults value for Day1-Day7 in sequence are: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.
NumericConvCode	String is a list of five characters to return when a numericconvcode command is processed in the PCE script. These represent the regional number punctuations for period, comma, plus sign, minus sign and space. The defaults are “.,+- ”. Where used, the parameter must be a single string representing each character in the sequence listed and without additional spaces.
NumericPadding	Char is a character to be returned when a numericpadding command is processed in the PCE script. This should be a single printable character.
UserValue1…16	Number is the value to return when a uservalue command is processed in the PCE script. The value of UserValue1 will be returned where the script parameter is 1, UserValue2 when the parameter is 2 and so on. By default 0 (zero) is returned for all keywords.

Example:

```
;Preferences0 defaults (English) are fine. Define 1 user value
<Preferences0>
UserValue1=10

;Preferences1 definition handles French
<Preferences1>
MonthName1=Janvier
Monthname2=Fevrier
;***etc.***
MonthsAbbrev1=Jan
MonthsAbbrev2=Fev
;***etc.***
Day1=Dimanche
Day2=Lundi
;***etc.***
NumericConvCode=".,+-_"
NumericPadding="."
```

<PrintDevice>

This section allows you to indicate the type of output datastream to be processed by a PCE script.

Syntax and defaults:

```
<PrintDevice>
PrintStream={AFPDS|POSTSCRIPT|NONE} ;default AFPDS
Resolution=Dpi ;defaults vary
PCC = {ANSI|ASCII|MACHINE|NONE} ;default ANSI
```

Keywords and parameters:

PrintStream	Indicates the type of output datastream to be processed. If you specify NONE no datastream output is expected. This option can be used when a PCE script has no datastream inputs or outputs to process. When specifying NONE you must set the Resolution keyword value to 0.
Resolution	You will need to code this keyword if the resolution of the output datastreams to be processed by PCE do not conform to the default Generate settings. These are: AFPDS – 240 PostScript – 72 . If you choose to use this keyword you must code the same parameter that was used when Generate originally created the output datastream(s) being manipulated – i.e. the default or the Resolution setting specified in the output device used when publishing a job.
PCC	Indicates the coding system to be used when inserting printer carriage control (PCC) bytes into a datastream if new presentation objects are being created.

Example:

```
<PrintDevice>
PrintStream=AFPDS
Resolution=1440
PCC=MACHINE
```

<System>

This section specifies settings that govern the way in which some PCE internal functions operate. Only include this section if you need to customize the standard system set-up; in most situations the default settings provide the optimal environment.

Syntax and defaults:

```
<System>
AsciiToAfpds=TableRef ;default 27
EbcdicToAfpds=TableRef ;default 0 (no translation)
AsciiToMetacode=TableRef ;default 20
EbcdicToMetacode=TableRef ;default 16
AsciiToPostscript=TableRef ;default 0 (no translation)
EbcdicToPostscript=TableRef ;default 0 (no translation)
AsciiToPdf=TableRef ;default 0 (no translation)
EbcdicToPdf=TableRef ;default 4
AsciiToVps=TableRef ;default 0 (no translation)
EbcdicToVps=TableRef ;default 0 (no translation)
PostscriptToHost=TableRef ;default 0 (no translation)
ResPackToHost=TableRef ;default 3 for EBSCDIC platforms, 0 for others
SystemCodePage=Integer ;default 37
TextSubsMethod=HASH|VSAM ;default HASH
VSAMKeyLength=Integer ;default 10
ConsolidateFonts={YES|NO} ;default YES
```

Keywords and parameters:

TableRef

Is a reference to the sequence number of a table within a Generate translation tables file (as assigned in the <Files> section). A setting of 0 (zero) indicates no translation should occur. You should refer to Precisely Support for information about the most suitable table for your environment. A standard translation tables file is provided with product distribution material. Custom files may be supplied in some circumstances.

ASCII/EBCDIC…

These keywords allow you to adjust the translation table that is used when creating new text elements within output datastreams. You may need to use these options if you are creating text that includes characters that are not typically available to fonts based on international English. You should use the keyword that matches your configuration of platform and output datastream; for example, use the AsciiToAfpds keyword if PCE is to run under Windows or Unix and the output datastream to be manipulated is AFPDS or EbcdicToPostScript when manipulating PostScript on an z/OS

platform. See TableRef above for information about the parameter.

PostscriptToHost	If this option is other than 0 (zero), PostScript output will have an additional translation applied before being output. This is typically used when running PCE under z/OS and where PostScript output is required to be printed/presented directly in an ASCII based environment. If the default is used PostScript commands are produced in a text format suitable for reading on the actual production platform. See TableRef above for information about the parameter.
ResPackToHost	In some scenarios parts of the HIP file need to be translated so that textual data it contains is suitable for the production host. The amount and relevance of textual data will depend on the output datastream for which it is intended. Most streams require only binary data and the HIP file is therefore not translated, but this can particularly be an issue with Postscript fonts when the production environment is EBCDIC based. **Integer** is the index number of the table in the Translation Tables file to be used for such translations.
SystemCodePage	Integer is the number of a host code page to be used instead of the default code page – US (37).
TextSubsMethod	Indicates the method used for reading a Generate lookup table (also known as a text substitution file) if the mapp command is used in the PCE script. Options are: HASH – text substitution strings are held in a text file. VSAM – text substitution strings are held in a VSAM structure (valid for z/OS systems only). If you code VSAM you should also code the VsamKeyLength keyword. See the mapp section of the **PCE command reference** on page 78 for details of text substitution file formats.
VsamKeyLength	Integer indicates the length of the key field used to contain the lookup labels. It is only used if the lookup table file is a VSAM structure otherwise it is ignored.
ConsolidateFonts	With the default setting **Yes** PCE will interpret each line in a file to determine if there is a font reference that needs remapping. **No** will not interpret each line and can result in faster PCE processing times.

Example:

```
<System>
EbcdicToAFPDS=0
TextSubsMethod=VSAM
VSAMKeyLength=16
ConsolidateFonts=NO
```

Start the job

The following pages provide a reference for all available INI sections and their associated keywords.

Starting the job

The following pages detail the start-up methods for PCE on all supported platforms.

Return Codes

DOC1PCE always returns 0 (zero) for successful completion or where warning messages (only) have been issued. Return code 16 is issued where a failure has occurred – i.e. where an abort message has been issued.

DOC1PCE under z/OS

Preparation:	PCE is submitted to the system via standard JCL. The standard file requirements for PCE are normally assigned by DD statements in the JCL. However, the actual DD labels required are defined within the PCE initialization file. Files specific to the application are defined within the PCE script file. They can be specified either as fully qualified dataset names or as a reference to DD labels used in PCE start-up JCL. Note that the PCE script file language does not provide a mechanism for specifying dataset attributes so where a new output file is required a DD reference should normally be used. No two files to be written by PCE (including the log and trace files) should be members of the same dataset.

EXEC card syntax:

```
EXEC PGM=DOC1PCE,PARM=('INI=DD:IniDD [symbol=val,symbol=val...]')
```

Parameters:

IniDD	is the DD label in the JCL that indicates the PCE INI file to be used.
symbol=val	indicates a value to be used to replace a symbol in the INI file. This is optional and may be repeated as many times as required.

Examples:

Specifying initialization file:

```
//DOC1PCE EXEC PGM=DOC1PCE,PARM='INI=DD:DOC1INIT'
```

Specifying initialization file and symbols:

```
//DOC1PCE EXEC PGM=DOC1PCE,PARM=('INI=DD:DOC1INIT Prefix=APP1,X=1')
```

Sample JCL

```
//Jobname JOB (xxx) ...(Rest of Job Card parms)
//DOC1PCE EXEC PGM=DOC1PCE,
//   PARM='/INI=DD:PCEINI'
//*Load lib for PCE + run-time libs if req’d
//STEPLIB  DD DISP=SHR,DSN=PROD.GEN.LOAD
//*PCE Script file
//PCECONT  DD DISP=SHR,DSN=PROD.GEN.CNTL
//*Initialization file (DD referenced in EXEC)
//PCEINI   DD DISP=SHR,DSN=PROD.GEN.RUN(A1INI)
//*Lookup tables file (DD referenced in INI)
//DOC1TSUB DD DISP=SHR,DSN=PROD.GEN.RUN(A1TSUB)
//*Translation tables file (DD referenced in INI)
//DOC1TTAB DD DISP=SHR,DSN=PROD.GEN.RES(DOC1TT)
//*GENERATE messages file (DD referenced in INI)
//DOC1MSG  DD DISP=SHR,DSN=PROD.GEN.MSG(MESSAGES)
//*Trace file (DD referenced in INI)
//EMFETRAC DD SYSOUT=C
//*Log file (DD referenced in INI)
//EMFELOG  DD DISP=SHR,DSN=PROD.GEN.LOG
//*Datastream input (DD referenced in script)
//PRININ   DD DISP=SHR,DSN=PROD.GEN.PCEHOLD(AFP19)
//*Journal relating to PRININ (DD referenced in script)
//DOC1JRN1 DD DISP=SHR,DSN=PROD.GEN.RUN(A1J1)
//*Datastream outputs (DDs referenced in script)
//PRINOUT1 DD SYSOUT=X,DCB=LRECL=8205
//PRINOUT2 DD SYSOUT=X,DCB=LRECL=8205
//*
```

DOC1PCE under Windows and UNIX

Preparation:

PCE is executed as a batch program started via the native command line of the relevant operating system interface, e.g. under Windows NT, type the start-up command in a command prompt window.

All files other than the initialization file itself are specified directly as part of the initialization file format or within the PCE script file being used.

Syntax:

```
doc1pce ini=IniFile [symbol=value symbol=value ...]
```

Parameters:

IniFile

identifies the path/filename of the PCE INI file to be used.

symbol=value

indicates a value to be used to replace a symbol in the INI file. This is optional and may be repeated as many times as required.

Notes:

The start-up syntax assumes that the DOC1PCE program is in the current ‘directory’ or ‘path list’ as these terms relate to the appropriate operating system.

Examples:

For Windows:

```
C:\DOC1HOST\RUN\DOC1PCE INI=C:\DOC1INI\JOB1.INI PATH=JOB1 X=1
```

For UNIX:

```
/doc1host/run/doc1pce ini=/doc1ini/job1.ini path=job1 x=1
```

7 - Defining external keyed images

The keyed object feature allows you to select some types of presentation objects dynamically according to variable criteria or ‘keys’. When you are working with keyed image resources you can import the relevant files into Designer where they can be referenced directly in a key map and be included in a HIP file ready for use with Generate.

It is not always practical to import all the required image files into the Designer however, and to deal with this scenario Designer & Generate support an external key map.

Note that you can also create a user exit program that directly returns the image to be placed when a key is used. See “Coding a Key Map user exit” on page 257 for details

The images you reference using an external key map can optionally be copied from a defined location and embedded in the output datastreams at production run-time. Where this is not the case the resources are assumed to be printer resident and only the image references are included in the datastream.

In this section

Embedding external keyed images

When embedding external keyed images no conversion takes place so you must ensure that the format of the images to be used is already compatible with the output datastreams being generated. The following table shows which image formats are supported for each datastream.

Output device	Supported resource format
AFPDS	Page Segment (FS45 & FS10 IOCA) and for printers with the relevant object container support, JPG, EPS, GIF, PDF Page Objects, TIF, BMP
PCL	PCL bmp – note that this is a proprietary format which is not generally available. Please contact Precisely Support for more information.
Postscript	EPS
Line Data	Not applicable
HTML	Any, all images are referenced
PDF	JPG and BMP – note that CMYK JPGs generated by products such as Adobe Photoshop may have had their color inverted so you must invert the colors of the image again using a similar product. scr 70619 - this anomally is now handled correctly in Designer (using Imagemagik), just not here
RTF	PNG

External key map file

An external key map is an XML file conforming to a predefined construct. It provides information about the keys to be used, the images associated with each key and information about image attributes.

An external key map must directly reference one or more output files by the reference name they are given when a publication is published in the Designer. If the key map is intended for use with publications generating multiple outputs you can associate different images to a single key – one for each output type.

images/KeyedObjectsOutputFiles.png

- Use the References in the Name column

The external key map also supports a **Wildcard** option that allows you to reference multiple images using a single entry. Where this option is used the images must be in the same location and have the same attributes – size, type etc.

The key map file can either be created manually or by a script you write yourself, or you can use the DOC1MAKE utility. DOC1MAKE interrogates the resources that are identified, extracts the required attributes from images and generates a key map file based on this information.

DOC1MAKE

Running DOC1MAKE (Windows only)

Preparation:

DOC1MAKE is provided with Designer distribution material and is executed from a Windows command prompt.

This utility is only available for Windows. If this is not your production platform you will need to transfer the resources to a location available to the utility. Where this is not possible consider creating the key map file manually perhaps using the wildcard option where many similar images are involved.

DOC1MAKE has multiple modes of operation:

Mode D is the all-inclusive method and directly generates an external key map from all the images stored in a defined directory. The keys and the image names are based on the file names within the directory with any extension removed. You may need to edit these later if the default names do not match the keys specified in the Designer or the actual file names on your production platform.

You may need to use the other modes where you cannot build the key map in a single operation or where you need to append information about new resources to an existing map.

Mode I appends an entry to an intermediate key map (which is created if necessary) for a single image file.

Mode X creates an external key map file from an existing intermediate file.

Mode S generates a schema for an existing external key map file.

DOC1MAKE and HTML output

Using the /H parameter ensures that keymap's resourcename is identical image's filename and is compatible for use with HTML output. If this parameter is not used, the image's filetype is omitted from the filename. Refer to the /H definition below for detailed information.

Syntax:

```
DOC1MAKE /M=D Input Output

/N=OutputName [/R=dpi /T=Type /E=EmbedFlag /D=DisposalOption /S=Schema
 /H /P=FormSpace /ecp=ECPFile]

DOC1MAKE /M=I Input Output
/N=OutputName /K=Key [/R=dpi /T=Type /I=Name /F=Filename /E=EmbedFlag
/D=DisposalOption /H /P=FormSpace /C=Delimiter /ecp=ECPFile]

DOC1MAKE /M=X Input Output
[/C=Delimiter /S=Schema /P=FormSpace /ecp=ECPFile]

DOC1MAKE /M=S [/S=Output /P=FormSpace /ecp=ECPFile]
```

Keywords & Parameters:

/M=D	Create external key map directly: Input is a directory containing image files from which an external key map is to be created. Output is the path/file name of an external key map (XML) which is created if it does not exist.
/M=I	Create intermediate key map: Input is the path/file name of a single image file. Output is an intermediate key map file to which an entry will be appended for the image and which is created if it does not exist.
/M=X	Create external key map from intermediate file: Input is an intermediate key map file. Output is an external key map file (XML) which is created if it does not exist.
/M=S	/S see below.
/N	Specifies the OutputName with which the key(s) will be used. See **“External key map file” on page 200** for details.
/K	Specifies the Key to be associated with an image.
/R	dpi specifies a resolution for an image where this cannot be ascertained from the image itself.
/T	Specifies an image Type where this cannot be ascertained from the image itself. See **XML structure of external key map** on page 227 for details of supported types and keywords.
/I	Specifies the Name by which the image is to be referenced in the output datastream where this is different from the base name of the input file.

/F	Filename is the path/file name of the image on the production system where required. By default, the input file name at the current location is assumed.
/E	If EmbedFlag = Y the image will be embedded in the output datastream when referenced by a publication. EmbedFlag = N (the default) indicates the image will not be embedded (i.e. it is printer resident).
/D	DisposalOption can be one of: 0 = Generate should retain the image in memory after use (default) 1 = free the image from memory as soon as it has been used 2 = free the image from memory when the Generate job is finished.
/H	Used when working with HTML output: • where the resource name must be the same as the physical filename • and where the resource name uses the exact case of the image filename and file type instead of the standard capitalized reference. Note that the /H parameter is applicable for use when DOC1MAKE is run in Directory (\M=D) and Intermediate (/M=I) modes only.
/P	FormSpace is used to indicate that EPS images that should be placed in the Form Space structure at the beginning of the Postscript output datastream rather than being inlined within the actual document pages. This may significantly reduce the size of the output file but as such images are referenced in printer memory be aware of constraints on your intended output device when deciding which images to include. Can be: TRUE = all EPS images are processed by this command) FALSE = no EPS images are processed by this command.
/C	Delimiter specifies the character to be used to separate entries in an intermediate key map file. You may need to specify this if the default (comma ‘,’) clashes with the kay map content
/S	Schema is the path and filename of an XML schema to be created or to be referenced (depending on the mode of operation).
/ecp	ECPFile is an ecp file containing the DBCS code pages required to support an RPU job on a non-ansi platform, such as in a non-Western production environment.

Example:

Generate a key map “keys.xml” for all the images in directory “.\graphics” for use when Generate is creating an output file with reference name “Output1”:

```
doc1make /M=D /E=Y graphics keys.xml /N=Output1
```

Add an entry for image "blackcat.bmp" with key “key1” to the intermediate file "temp.txt":

```
doc1make /M=I blackcat.bmp temp.txt /K=key1 /N=Output1
```

Generate a key map “keys.xml” from the intermediate file “temp.txt”:

```
doc1make /M=X temp.txt keys.xml
```

Copy the XML key map schema to “kms.xsd” in the \temp directory:

```
doc1make /M=S /S=c:\temp\kms.xsd
```

XML structure of external key map

All parameters are specified as quoted strings. You may include other valid XML statements within the structure if desired.

<ExternalKeyedImages> top level structure

Structure

```
<ExternalKeyedImages>
 Label=string
 Version=string
 <ImageDefaults>
 </ImageDefaults>
 <Keys>
  <KeyEntry Key>
   <Image>
    <ImageDeviceInfo>
    </ImageDeviceInfo>
    ...
   </Image>
   <Image>
   ...
  </KeyEntry>
  ...
 </Keys>
</ExternalKeyedImages>
```

Attributes

Label reference name of the file

Version the XML file version (initially version 1).

Comments

Parent: None

Contains: <ImageDefaults>, <Keys>

<ImageDefaults> section

Function

This section specifies default attributes for all images referenced in the key map. These will be used when the relevant attribute is not specified for an individual image. Note that all the attributes are required.

Structure

```
<ImageDefaults>
 Width=string
 Height=string
 Resolution=string
 Embed=string
 Disposal=string
</ImageDefaults>
```

Attributes

Width width of the image in pixels

Height height of the image in pixels

Resolution resolution of the image in pixels per inch

Embed the image can be embedded in the output datastream

True – embed the image

False – don't embed the image

The option to embed an image varies depending on the output device you are using as follows:

AFPDS – True or False allowed.

Postscript – True or False allowed.

HTML – False only allowed.

PDF – True only allowed.

RTF – True only allowed.

Disposal this option indicates how Generate should manage the system memory associated with the image files. Retaining images in memory may allow Generate to run significantly faster but where the same external images are used repeatedly in an application but this may impose an unacceptable burden on system resources where many or large images are involved. Can be: **Retain** – keep images in memory indefinitely. **ClearAfterUse** – free memory as soon as an image has been used. **ClearAfterJob** – free memory at the end of the job.

Comments

Parent: <ExternalKeyedImages>

Example

```
<ImageDefaults
 Width="100"
 Height="100"
 Resolution="72"
 Embed="true"
 Disposal="Retain" />
</ImageDefaults>
```

<Keys> section

Function

The section is a container for all <KeyEntry> sections. It has no keywords.

Structure

```
<Keys>
 <KeyEntry>
  ...
 </KeyEntry>
</Keys>
```

Attributes

None

Comments

Parent: <ExternalKeyedImages>

Contains: <KeyEntry> (at least one)

Example

See <KeyEntry>

<Image>

Function

An image referenced by a key. You may want to add several images, typically each one for a different output device.

Structure

```
<Image>
 Width=string
 Height=string
 Resolution=string
 <ImageDeviceInfo>
  ...
 </ImageDeviceInfo
</Image>
```

Attributes

Width width of the image in pixels

Height height of the image in pixels

Resolution resolution of the image in pixels per inch.

Comments

Parent: KeyEntry

Contents: ImageDeviceInfo (any number, but at least one)

The image dimensions are only used at design time to place the image on the page and flow text around it. If the dimensions of the actual images referred to by the key are different then the behavior in Generate will be unpredictable, for example, the driver may overprint.

Example

```
<Image>
 Width="100"
 Height="120"
 Resolution="72"
 <ImageDeviceInfo>
  ...
 </ImageDeviceInfo>
</Image>
```

<ImageDeviceInfo>

Function

Information about the image for the output device.

Structure

```
<ImageDeviceInfo>
 Device=string
 ResourceName=string
 FileName=string
 ImageType=string
 Embed=string
 Disposal=string
 PDFPixelHeight=string
 PDFPixelWidth=string
 FormSpace=string
</ImageDeviceInfo>
```

Attributes

Device this attribute is compulsory and indicates the reference name of the output file with which the key will be used. See **"External key map file" on page 200** for details.

ResourceName this attribute is compulsory and indicates the name of the image in the output stream. For multiple images, where the name includes the key, use the wildcard character to represent the key in the base part of the image name.

FileName this attribute is compulsory and indicates the path and name of the image file (including extension if appropriate). For multiple images, where the name includes the key, use the wildcard character to represent the key in the base part of the image name.

ImageType this attribute is compulsory and indicates the image format – can be:

PSG – AFP bitmap

F45 – AFP IOCA FS45

BMP – Bitmap

JPG – Joint Photographic Experts Group Format

GIF – Graphical Interchange Format

TIF – Tagged Interchange Format

PNG – Portable Network Graphics

EPS – Encapsulated Postscript image

PS – For PostScript output this indicates that only a reference to an image resource should be placed in the stream. The image type is undefined. It is the users responsibility to ensure the relevant resource is available to the target output environment and is compatible with the output type and document designs.

Embed optional. Refer to **<ImageDefaults> section** on page 229 for details.

Disposal optional. Refer to **<ImageDefaults> section** on page 229 for details.

PDFPixelHeight the height of the image in pixels. This option is only valid when the **ImageType**=JPG and the intended output device is PDF.

PDFPixelWidth the width of the image in pixels. This option is only valid when the **ImageType**=JPG and the intended output device is PDF.

FormSpace indicates that an EPS image that should be placed in the Form Space structure at the beginning of Postscript output datastream rather than being included within the actual document pages. This may significantly reduce the size of the output file but as such images are referenced in printer memory be aware of constraints on your intended output device when deciding which images to include: **True** – place image in form space **False** – don’t place image in form space This option is only valid when the ImageType=EPS and the intended output device is Postscript.

ColorSpace defines the output color space for JPEG images, omitting this value may produce invalid PDF. This option is only valid when the **ImageType=JPG** and the intended output device is PDF.

- Attribute values can be:
  - Greyscale – value of `2`
  - CMYK – value of `12`
  - RGB - value of `1`
  - Undefined – value of `0`

Comments

Parent: Image

Contents: None

Example

```
<ImageDeviceInfo>
 Device="Output1"
 Embed="true"
 Disposal="Retain"
 ImageType="PSG"
 ResourceName="rn1"
 FileName="/usr/home/images/myimage1.psg"
</ImageDeviceInfo>
<ImageDeviceInfo>
 Device="Output2"
 Embed="true"
 Disposal="Retain"
 ImageType="F45"
 ResourceName="img%"
 FileName="C:\Customers\KeyIms\AFP\img%.F45"
</ImageDeviceInfo>
```

Example Keyed image XML

This example of XML for external keyed images has three keys with images defined for different devices.

```
<?xml version="1.0" encoding="UTF-8"?>
<ExternalKeyedImages Label="MyFile" Version="1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="extkeyim.xsd"
   <ImageDefaults Width="100" Height="100" Resolution="72" Embed="true"
 Disposal="Retain" />
   <Keys>
      <KeyEntry Key="100101">
         <Image Width="123" Height="123" Resolution="72">
            <ImageDeviceInfo
               Device="Output1"
               Embed="true"
               Disposal="Retain"
               ImageType="PSG"
               ResourceName="rn1"
               FileName="/usr/home/images/myimage1.psg"/>
            <ImageDeviceInfo
               Device="Output2"
               Embed="true"
               Disposal="Retain"
               ImageType="BMP"
               ResourceName="rn2"
               FileName="/usr/home/images/myimage1.bmp"/>
          </Image>
        </KeyEntry>

        <KeyEntry Key="100102">
            <Image Width="456" Height="456" Resolution="96">
               <ImageDeviceInfo
                  Device="Output3"
                  Embed="false"
                  Disposal="Retain"
                  ImageType="EPS"
                  ResourceName="rn3"
                  FileName="/usr/home/images/myimage3.eps"/>
                  FormSpace="TRUE"
               <ImageDeviceInfo
                  Device="Output4"
                  Embed="true"
                  Disposal="ClearAfterUse"
                  ImageType="BMP"
                  ResourceName="rn4"
                  FileName="/usr/home/images/myimage3.bmp"/>
               </Image>
            </KeyEntry>

            <KeyEntry Key="100103">
               <Image>
                  <ImageDeviceInfo
                     Device="Output5"
                     ImageType="EPS"
                     ResourceName="rn5"
                     FileName="/usr/home/images/rainbow.eps"/>
               </Image>
            </KeyEntry>
         </Keys>
      </ExternalKeyedImages>
```

Note:

- A key can have serveral images, each for a different output device.
- The size of the actual images must match the dimensions defined in the image attributes,for example:

  ```
  <Image Width="456" Height="456" Resolution="96">
  ```

- The Device name must match the name defined in the Publish Wizard Input/Output section, for example:

  ```
  Device="Output4"
  ```

This XML example defines multiple images.

```
<?xml version="1.0" encoding="UTF-8"?>
<ExternalKeyedImages Label="MyFile" Version="1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="extkeyim.xsd"
<ImageDefaults Width="1240" Height="500" Resolution="600" Embed="false"
 Disposal="Retain"
/>
   <Keys>
      <KeyEntry Wildcard="%">
         <Image>
            <ImageDeviceInfo
               Device="Output1"
               ImageType="BMP"
               ResourceName="img%sm"
               FileName="/usr/home/images/img%sm.bmp"/>
        </Image>
      </KeyEntry>
   </Keys>
</ExternalKeyedImages>
```

Note: The & symbol in img%sm.bmp will be substituted with the key value at run time.

8 - Working with resources in a HIP file

Along with all the other resources required to process a publication in the production environment, a HIP file contains all the font and image resource files required actually to print or present the output datastreams it produces.

Such resources will automatically be embedded into the output datastream(s) produced by Generate unless your intended output datastream does not support resource embedding or you explicitly override embedding when publishing a publication.

In this section

Extracting and manipulating resources

The DOC1RPU utility provides the ability to extract and manipulate resources within a HIP file. You may need to use it in the following situations:

Extracting resources You may want to extract the resources into separate files and then pass them independently to your output device environment. This is often done to reduce the size of output datastream files or to improve printer performance where the datastream supports the use of external resources.

Resident resources If you are extracting resources as above or if the intended output device already has access to all the resource files referenced by the publication you may want to indicate to Generate that specific resources are resident and therefore do not need to be embedded in the output datastream.

Customizing the use of external documents for PostScript output If a publication design contains external documents and you are generating PostScript these will have been created as EPS resources within the HIP. You may want to specify how such resources are used within the actual output datastream (in the Form Space file header or inline within the actual document pages) as this may significantly affect the size of the output file.

Customizing PostScript for Xerox devices In some Xerox environments you may want to indicate that resources are to be deleted from printer memory after they have been used.

Support for OnDemand DBCS fonts

AFP DBCS fonts produced by Generate are based on a custom code page. Fonts for use with IBM OnDemand must conform to a specific code page or must be mapped using code page information in the HIP file. This information can then be integrated into existing configuration files in the OnDemand environment.

DOC1ACU (AFP Codepage Utility) reads the HIP file and creates the necessary code page information to be integrated into IBM OnDemand for viewing the generated AFP. Note that the DOC1ACU program runs in a Windows environment only.

RPU

RPU is controlled by an initialization file (INI) which can be prepared using any standard text editor. Details for coding the INI and instructions for starting the DOC1RPU program are provided in the sections that follow.

You must always specify the InputHip keyword to indicate the HIP file to be processed. If you are adjusting the settings of resources within the HIP you may also want to specify the OutputHip keyword to receive the amended file.

As a HIP file can contain resources for several output devices you must always code the ResourceType and Resolution keywords in the <OutputDevice> section of the INI to identify the specific set of resource files to be used.

When extracting resources they are normally output as individual files to the locations specified by the …Mask keywords in the <Files> section of the INI. However Note that if OutputStream is coded all …Mask options are ignored.

RPU initialization file reference

RPU initialization file format

	The INI consists of keywords and parameters coded within sections. The sections must be introduced with the relevant string within angle brackets, for instance: `<Files>`. Parameters are normally coded as literal values but can be specified dynamically if required by defining them wholly or partly as symbols. Values can be assigned to such symbols when starting the DOC1RPU program. Within the INI file symbols are referenced by coding the symbol name to be used within percentage signs. For example: `InputHip=%HipName%`

Syntax:

```
<Files>
Messages=Filename
InputHip=Filename
OutputHip=Filename.
OutputStream=Filename
FontMask=location
CharSetMask=location
CodePageMask=location
ImageMask=location
OverlayMask=location
DOC1ecp=Filename
```

```
<OutputDevice>
ResourceType={AFP|HTML|POSTSCRIPT|PDF} ;default AFP
Resolution=dpi ;default 240
Format={STANDARD|BARR400|BARRPC|CRLF|KSDSAFP|LINE|PP4235|RDW|RECORD|RRDSAFP|
RRDSMTC|SPUR|VSAMAFP|WSAFP|FormatParms}
PCC = {ANSI|ASCII|MACHINE|NONE} ;default ANSI
```

```
<Commands>
Extract={ALL|NONE|ResName} ;default ALL
MarkResident={ALL|NONE|ResName} ;default ALL
List={ALL|NONE|ResName} ;default ALL
Delete={ALL|NONE|ResName} ;default NONE
DeleteAfterExtract={TRUE|FALSE} ;default FALSE
```

```
<AFP>
UsePageSegment = {TRUE|FALSE};default FALSE
```

```
<PostScript>
MarkForFormSpace={ALL|NONE|ResName} ;default NONE
MarkFormForDeletion={ALL|NONE|ResName} ;default ALL
UnmarkFormForDeletion={ALL|NONE|ResName} ;default NONE
```

```
<System>
SystemCodePage=Integer ;default 37
OutputDataFormat={STREAM|RECORD} ;default (Platform dependent)
```

Data types:

Filename	is a path/file name or label conforming to the conventions of the operating system that will run RPU.

Location

is a reference to a file location in the format required by the host operating system. For Path parameters the wildcard (%1) should be included to indicate the position of the actual filenames in the path structure.

Examples:

Windows – `c:\gen\prtfiles\%1.FNT`

UNIX – `/gent/prtfiles/%1.FNT`

z/OS – `DD:FNTFILES(%1)`

dpi

an integer indicating an output device resolution, e.g 72, 96, 240, 300 etc.

TString

is a string of alphanumeric characters or their representation as in hexadecimal notation (e.g. "X'A1D390B2B1'") depending on the option selected.

ResName

identifies a resource known to be present in the HIP file. Specify the name of the resource as it known to the Designer without extension.

String

is a string of alphabetic characters enclosed in quotes.

KeywordList

is a comma separated list of keywords.

Using ALL, NONE or resource names

Several keywords have these options.

ALL – all resources are processed by this command

NONE – no resources are processed by this command

ResName – identifies a specific resource within the Pack to be processed. Use the List option to establish the required resource names if required.

> **Note:** If you are specifying individual resources you may code the command as many times as required.

Keywords and parameters:

<Files>

Messages

FileName is the utilities message text file. You must specify the Messages file provided for use with your current version of the Generate production environment. Where possible code this keyword as the first entry in the INI.

InputHip

FileName identifies an existing Generate HIP file on which RPU will act.

OutputHip

If RPU is to modify the contents of the HIP file, FileName identifies the file which will receive the modified data. If you do not specify this keyword the file specified as InputHip will be modified.

OutputStream	Where it is possible for the extracted resources to be used as a single stream (e.g. not for AFP environments) use this keyword to specify the FileName to receive the merged resources. If you omit this keyword the resources will be extracted as separate files in the appropriate …Mask locations.
FontMask	When extracting resources to individual files this keyword specifies a location to receive font resources. For AFP resources this location will always be used for coded font files. It will also be used for character sets and code pages unless the CharSetMask and CodePageMask keywords are coded. For all other resource types all font files are created at this location.
CharSetMask	When extracting AFP resources to individual files this keyword specifies a location to receive character set resources. If it is not coded such resources will be created at the FontMask location.
CodePageMask	When extracting AFP resources to individual files this keyword specifies a location to receive code page resources. If it is not coded such resources will be created at the FontMask location.
ImageMask	When extracting resources to individual files this keyword specifies a location to receive image resources.
OverlayMask	When extracting AFP resources to individual files this keyword specifies a location to receive overlay resources.
DOC1ecp	FileName is the DOC1ecp file containing the DBCS code pages required to support a RPU job on a non-ansi platform is located, such as in a non-Western production environment.
<OutputDevice>	
ResourceType	Indicates the type of output device for which resources are to be extracted.
Resolution	dpi indicates the resolution of resources to be extracted. This works in conjunction with ResourceType to identify the required set of resources within the HIP.
Format	If the resources are being output as a stream this optional parameter allows you to define the structure of its logical records. You will need to use this option if the default record structure generated by Generate is not suitable for your requirements. The defaults are: Metacode on all platforms other than z/OS – RDW All others – STANDARD Refer to **Output datastream formats** on page 336 for a detailed discussion about logical records and formatting options including examples.
PCC	If the resources are being output as a stream this optional parameter to define the coding system to be applied to the printer carriage control (PCC) byte if this is being used in the output. Options are: ANSI ASCII MACHINE
<Commands>	

Extract	Use this keyword to indicate what resources (if any) you require to be extracted from the HIP. Options: see **Using ALL, NONE or resource names**.
MarkResident	Use this keyword to indicate what resources (if any) you require to be marked as resident within the Pack. When this is done a flag is set within the Pack to inform Generate to not embed such resource in the output datastream when the Pack is used by the main application. If you are marking individual resources you may code MarkResident as many time as required. Options: see **Using ALL, NONE or resource names**.
List	RPU can output the names of the resources within the Pack file along with attribute information. Use this keyword to indicate if this option is required. If you are Listing individual resources you may code List as many time as required. Options: see **Using ALL, NONE or resource names**.
Delete	Use this keyword to indicate what resources (if any) you require to be permanently deleted from the Pack. If you are deleting individual resources you may code Delete as many time as required. Options: see **Using ALL, NONE or resource names**.
DeleteAfterExtract	If TRUE all resources identified in Extract keywords will also be deleted from the pack once they have been extracted. Note that if Delete keywords are specified they may override this setting.
<AFP>	
UsePageSegment	If TRUE FS45 and FS10 images, when extracted, will be within a page segment. Note that the default for this setting is FALSE, where FS45 and FS10 images are not within a page segment when extracted.
<PostScript>	
MarkForFormSpace	This keyword is only used if the HIP file contains EPS resources used for external documents in a PostScript environment. Use this keyword to indicate that the EPS resources that should be placed in the Form Space structure at the beginning of the PostScript output datastream rather than being inlined within the actual document pages. This may significantly reduce the size of the output file but as such resources are referenced in printer memory be aware of constraints on your intended output device when deciding which fonts to include. Options: see **Using ALL, NONE or resource names**.
MarkFormForDeletion	This keyword is used only in relation to PostScript Forms generated by Xerox Decomp Services. The resources indicated will be flagged for deletion from printer memory once the output datastream in which they are embedded has been printed. This is only relevant when the printer environment supports this concept. Options: see **Using ALL, NONE or resource names**.
UnmarkForm…	Where Xerox PostScript Forms within a HIP file have previously been marked for deletion using the keyword above you can use this keyword to reverse the setting, i.e. the resources will not be flagged for deletion within the output datastreams in which they are embedded. Options: see **Using ALL, NONE or resource names**.

<System>

SystemCodePage	Integer is the number of a host code page to be used instead of the default code page – US (37).
OutputDataFormat	Use this keyword to specify the type of output for your operating platform. Options: OS390 = RECORD, all other platforms = STREAM.

Example:

Extracting individual AFP resources under Windows, UNIX, etc.:

```
<Files>
Messages=messages.dat
InputHip=\gen\resources\jobin.hip
OutputHip=jobout.hip
FontMask=\afpres\codefont\%1.icf
CharSetMask=\afpres\charsets\%1.ibb
CodePageMask=\afpres\codepage\%1.icp
ImageMask=\afpres\pagesegs\%1.psg
OverlayMask=\afpres\overlays\%1.ovl<OutputDevice>
ResourceType=AFP
Resolution=300<Commands>
Extract=X0FONT01
Extract=X0FONT02
Extract=IMG01
MarkResident=ALL
DeleteAfterExtract=TRUE
```

Marking EPS resources so they are placed and referenced in the PostScript Form Space:

```
<Files>
Messages=messages.dat
InputHip=\gen\resources\jobin.hip
OutputHip=jobout.hip
<OutputDevice>
ResourceType=PostScript
Resolution=300
<PostScript>
MarkForFormSpace=IMGSIG1
MarkForFormSpace=IMGSIG2
MarkForFormSpace=IMGSIG3
MarkFormForDeletion=ALL
```

FS45 and FS10 images are placed within a page segment:

```
<Files>
Messages=messages.dat
InputHip=\gen\resources\jobin.hip
OutputHip=jobout.hip
<OutputDevice>
ResourceType=AFP
Resolution=300
<AFP>
UsePageSegment=TRUE
```

Running DOC1RPU

Running DOC1RPU under Windows and UNIX

Preparation:

Create an RPU INI file using a standard text editor such as Windows Notepad. See **RPU initialization file reference** on page 240.

DOC1RPU is run from the command line of an appropriate operating system window.

Syntax:

```
doc1rpu ini=RpuIni [symbol=value symbol=value ...]
```

Parameters:

ini	RpuIni is the path/filename of the RPU INI file to be used.
symbol=value	indicates a value to be used to replace a symbol in the INI file. This is optional and may be repeated as many times as required.

Example:

```
doc1rpu ini=\doc1host\run\rpuj1.ini Extract=ALL
```

Running DOC1RPU under z/OS

Preparation:

Create an RPU INI file using the standard text editor. See **"RPU initialization file format" on page 216**.

DOC1RPU is submitted using standard JCL. The DD statements required in the JCL will depend on the file related keywords specified in the INI.

EXEC card syntax:

```
EXEC PGM=DOC1RPU,PARM=('INI=DD:IniDD [,symbol=value,symbol=value,...]')
```

Parameters:

INI	IniDD is the DD label in the JCL that indicates the RPU INI to be used.
symbol=value	indicates a value to be used to replace a symbol in the INI file. This is optional and may be repeated as many times as required, each instance separated by a comma.

Examples:

```
//DOC1RPU EXEC PGM=DOC1RPU,PARM='INI=DD:DOC1INIT'
//DOC1RPU EXEC PGM=DOC1RPU,PARM=('INI=DD:DOC1INIT,EXTRACT=ALL')
```

Sample JCL:

```
//Jobname JOB (xxx) ...(Rest of Job Card parms)
//DOC1RPU EXEC PGM=DOC1RPU,
// PARM='/INI=DD:RPUINI'
//*Load lib for PCE + run-time libs if req’d
//STEPLIB DD DISP=SHR,DSN=PROD.GEN.LOAD
//*Initialization file (DD referenced in EXEC)
//RPUINI DD DISP=SHR,DSN=PROD.GEN.RUN(A1INI)
//*Generate messages file (DD referenced in INI)
//DOC1MSG DD DISP=SHR,DSN=PROD.GEN.MSG(MESSAGES)
//*Input HIP file (DD referenced in INI)
//INHIP DD DISP=SHR,DSN=PROD.GEN.HIPS(JOB10)
//*Output HIP file (DD referenced in INI)
//OUTHIP DD DISP=SHR,DSN=PROD.GEN.HIPS(JOB10A)
//*Dataset to contain extracted fonts (DD referenced in INI)
//FONTMASK DD DISP=SHR,DSN=PROD.GEN.HIPS
//*Other DDs may follow as required
//*
```

DOC1ACU

ACU provides the ability to extract configuration information required to view Generate AFP output using custom code pages in a third party viewer such as that provided in the IBM On Demand environment.

You must ensure that **Build fonts for IBM OnDemand** output device has been enabled for DOC1ACU to process the HIP file correctly. Refer to the Designer User’s Guide for further information.

ACU reads the HIP file and creates a number of configuration files as follows: coded.fnt – describes any DBCS fonts found in the HIP file, mapping the coded font to code page and character set identifiers found in cpdef.fnt and csdef.fnt. cpdef.fnt – maps the code page in the coded.fnt record to a Windows character set or custom character set. csdef.fnt – maps the character set in coded.fnt to the font family of the installed Windows font to be mapped to. 65280.ucm – the Unicode Character Map (UCM) file is based on code page identifiers from cpdef.fnt

The information in all the above files except 65280.ucm must be merged into corresponding OnDemand control files.

Note that DOC1ACU assumes that all font Typeface / language pairings use the same code page. Only fonts generated through the 'default' code page setting in the Resource Map associated to output device can be processed by DOC1ACU.

Instructions for starting the DOC1ACU program are provided in the sections that follow.

You must always specify the InputHip keyword to indicate the HIP file to be processed. When DOC1ACU is initially run it will create a configuration file specified by the ConfigFile keyword. This file contains code page and font information based on custom code pages found in the HIP file. Information held in this file is reused by ACU on subsequent runs.

Running DOC1ACU under Windows

Preparation: DOC1ACU is run from the command line in operating system window.

Syntax:

```
doc1acu ConfigFlie InputHip AFPResolution [Outputdir][doc1ecp]
```

Parameters:

ConfigFile	contains code page and font identification information. Existing values can be used by DOC1ACU, new values read from the HIP file
InputHip	identifies an existing Generate HIP file on which ACU will process.
AfpResolution	the resolution of the AFP output datastream as defined in the Generate production job. Specify 240, 300 or 600.
Outputdir	indicates the location for the output from ACU to be stored. The local directory is used if this keyword is not specified.
doc1ecp	is the path/file name of the Generate Extended Code Page file which will be required for most non-Western applications. The doc1ecp file in the local directory is used if this keyword is not specified.

Example:

```
doc1acu config.dat myafp.hip 300
```

9 - Processing PDF output

PDF output data can be created by Generate as a single compound stream containing all the documents processed. When this type of output datastream is initially created it is not readable by a PDF reader such as Acrobat Reader, nor can it be distributed by e-mail.

Note that if you are working in an e2 environment PDF output created by Generate can be loaded directly into the Document Repository.

In this section

DIME

DOC1DIME (Dynamic Internet Mailing Engine) is a utility that allows the extraction of individual documents from the compound PDF output produced by Generate. It can then either:

- write the extracted data as files to a specified location;
- send the documents using an API to a locally available e-mail client.

An initialization file (INI) is used to customize the required environment for a particular execution of DIME.

If the <Files>OutputFileMask keyword is not specified in the INI file DIME will assume that e-mailing is required. If you need to extract both documents and e-mail then you will need to run DIME twice.

You will also need to include your license keycode details for Generate in the <LicenseInfo> section of the INI file.

Extracting to File

If the `<Files>OutputFileMask` keyword is specified then DIME will assume that the extracted compound PDF documents are to be saved as individual files.

When using this option the OutputFileMask keyword specifies a template for the path/filenames that will contain the individual documents. The `%1` code in the parameter supplied indicates the position of file base names within the path structure. This will be replaced by a unique 7 character id (based on incremental hex values) for each extracted document. The `%2` code in the parameter acts the same as the %1 code, except that the unique 7 character id is based on decimal values.

You will need to specify the `%1/%2` in a path structure suitable for the operating system under which DIME is to run. On operating systems where file extensions are supported, the required extension must be consistent and must be coded as part of the reference. Under z/OS, `%1/%2` represents all but the first character of member names to be generated in a partitioned dataset which must be identified as a fully qualified name (not a DD label).

E-Mailing

The mailing function of DOC1DIME is available only under Windows. It uses one of two Windows APIs to interface to the e-mail client available on the system on which it is running: – Common Messaging Calls (CMC) – Messaging Application Programming Interface (MAPI)

The selected API must be installed and configured under Windows for the mailing feature to be used – refer to your Microsoft documentation for more information.

Note that if you are using the e-mail template option (defined as a document attribute in Designer), the MAPI API must be used

The PDF documents are despatched as attachments to the e-mails. The subject title and message used with mailings is customizable but is the same for every recipient of a particular DIME process. The e-mail address or address book identifier used for mailing must be embedded within the PDF file. Generate produces such addresses when the `Document Attribute/e-mail Address` object has been created within the appropriate Document Layout when designing the application using the Designer. You must therefore ensure that the object exists and has logic that will generate the required addresses. Refer to the DOC1User's Guide for more information.

The text of the e-mail message to which the PDF file is attached must be entered in a separate file which is then referenced in the Initialization File. The message file must contain unformatted ASCII text only. Other attributes of the e-mail – the subject title and the name used to reference the attached PDF – are defined directly in the INI file.

By default, DIME will assume a generic internet mail server is being used – i.e. the mail addresses have the format SMTP:address. If you are using a mail server that needs a specific address format you will need to code the `AddressMask` INI option to specify it. For example: `AddressMask=NOTES:%1` for a Lotus Notes mail server `AdressMask=EX:%1` for a Microsoft Exchange mail server.

DIME INI Reference

DIME initialization file format

The DIME INI file specifies the environment for a particular execution of the DOC1DIME PDF extract utility.

It contains four sections that must be introduced with the relevant string within angle brackets, for instance: `<Files>`. Each section can contain a range of keywords and their associated parameter as listed below.

Parameters are normally coded as literal values but can be specified dynamically if required by defining them wholly or partly as symbols. Values can be assigned to such symbols when starting the DOC1RPU program. Within the INI file symbols are referenced by coding the symbol name to be used within percentage signs. For example: `Input=%FName%`

If the <Files>OutputFileMask keyword is specified DIME will assume that the PDF documents are to be output as files. If it is not specified DIME assumes that the documents are to be e-mailed.

Syntax:

```
<Files>
Messages=Filename ;mandatory
Input=Filename ;mandatory
OutputFileMask=Location
LogInfo=Filename ;defaults to none
```

```
;This section is required only when e-mailing
<e-mail>
SubjectText="String"
AttachmentName="String"
API={CMC|MAPI} ;default CMC
UseAddressBook={YES|NO} ;default NO
AddressMask=AddressMask ;default SMTP:%1
ProcessingTrace={YES|NO} ;default NO
DeleteAfterSend={YES|NO} ;default NO
```

```
<Exception>
SuppressWarnMsg={YES|NO} ;default NO
SuppressInfoMsg={YES|NO} ;default NO
SuppressDumpMsg={YES|NO} ;default YES
OnMapiError={CONTINUE|STOP} ;default STOP
```

Data types:	
Filename	is a path/file name or label conforming to the convention required for the host operating system. The **Preface** on page 4 section of this manual contains guidelines for the expected naming conventions for all supported platforms.
Location	is a reference to a file location in the format required by the host operating system. This is a template for the path/filenames to be created by the extraction process. It which must always include a %1 code to indicate the position of file base names within the path structure. This will be replaced by a unique 7 digit name based an incremental hex values as the PDF documents are generated. You will need to specify the %1 in a path structure suitable for the operating system under which DIME is to run. On operating systems where file extensions are supported, the required extension must be consistent and must be coded as part of the reference. Under z/OS, %1 represents all but the first character of member names to be generated in an existing partitioned dataset which must be identified as a fully qualified name (not a DD label). Examples: Windows: `c:\gen\pdfout\%1.pdf` UNIX – `/gen/pdfout/%1.pdf` z/OS – `DD:PDFOUT(%1)`
String	is a string of alphanumeric characters

AddressMask	is a string which includes the code ‘%1’ which will be resolved with the existing e-mail address within the PDF documents.
Keywords and parameters:	
<Files>	
Messages	FileName is the diagnostic message text file. You must specify the Messages file provided for use with your current version of the Generate production environment. You are advised to code this keyword as the first entry in all Initialization Files as any errors found while processing the Initialization File itself can only be reported once the Diagnostic Messages File has been identified and loaded.
Input	FileName is a PDF file previously produced by Generate. Note that PDF produced by other applications is not supported by DIME.
OutputFileMask	For DIME only. If this keyword is coded DIME will assume that the documents extracted from a PDF file are to be written to individual files. If it is not coded DIME assumes that internet mailing is required. Location specifies a template for the path/filenames that will contain the individual documents. The %1 is replaced by an 8 digit value coded in hex. For example, %1 will be 00000001 for the first document, 0000000A for the 10th document, 0000001F for the 31st document and so on. The %2 code can be used in the parameter and acts the same as the %1 code, except that the unique 7 character id is based on decimal values. You must ensure that the template allows valid filenames to be generated.
LogInfo	If ProcessingTrace = YES, FileName will be appended with trace information from the current execution of DIME.
<E-mail>	
SubjectText	String is the text that will be used in the subject field of each mail message.
AttachmentName	String is the name by which the attached PDF document is referenced within the e-mail.
API	Indicates the type of API to be used calling e-mail functions.Options are: CMC – use the Common Messaging Calls API (this is the default) MAPI – use the Messaging Application Programming Interface API. This API must be used if you are using the e-mail template option.
UseAddressBook	If YES is specified, the address book available with the local e-mail client will be used to translate the address in the PDF document to an alias before it is despatched. If no address book is found this option is ignored.

AddressMask	Permits modifications to the e-mail addresses generated by Generate to reflect the mail server being used. The default is SMTP:%1 which is suitable for use with generic internet mail servers. You will need to specify a different value if you are using a specific mail server. Possible parameters include: `NOTES:%1` for Lotus Notes mail server `EX:%1` for Microsoft Exchange mail server.
ProcessingTrace	If YES is specified, messages received from processing e-mail will be written to file identified by Files/LogInfo if coded.
DeleteAfterSend	If YES e-mail will be deleted from the Outlook outbox after it is sent. If the default NO is specified the e-mail will remain in the outbox until deleted manually.
<Exception>	
SuppressWarnMsg	If YES is specified Generate will not issue any messages classed as warnings, i.e. those with suffix 'W'.
SuppressInfoMsg	If YES is specified Generate will not issue any messages classed as information, i.e. those with suffix 'I'.
SuppressDumpMsg	If NO is specified, Generate can issue a dump format that may help to isolate the application data record/field being processed at the time the warning was issued. Set this option to NO if you need to see such dumps.
OnMapiError	If the default STOP is specified then the e-mailing process will terminate when it encounters an invalid e-mail address. If CONTINUE is specified then the process will send e-mails to the valid addresses, ignoring the invalid addresses.

Example:

```
<Files>
Messages=/doc1host/messages.dat
Input= /doc1host/emfeout.pdf
LogInfo=/doc1host/pdftrail.txt

<e-mail>
SubjectText="Your statement for October"
AttachmentName="Oct2001"
API=MAPI
UseAddressBook=YES
AddressMask=NOTES:%1
ProcessingTrace=YES
DeleteAfterSend=YES

<Exception>
SuppressWarnMsg=YES
SuppressInfoMsg=YES
SuppressDumpMsg=NO
OnMapiError=CONTINUE
```

Running DOC1DIME

DOC1DIME under Windows and UNIX

Purpose:	Extracts individual PDF documents from a compound printstream file created by Generate and, optionally for Windows only, mails them to appropriate e-mail addresses.
Preparation:	DOC1DIME is run from the command line of an appropriate operating system window.

Syntax:

```
doc1dime ini=IniFile [symbol=value symbol=value ...]
```

Parameters:

IniFile	is the path/filename of the DIME INI file to be used.
symbol=value	indicates a value to be used to replace a symbol in the INI file. This is optional and may be repeated as many times as required.

Example:

```
doc1dime ini=\doc1host\run\dimej1.ini val1=earlyrun
```

DOC1DIME under z/OS

Purpose:	Extracts individual PDF documents from a compound printstream file created by Generate and stores them in separate files.
Preparation:	PDF documents are extracted as the members of a partitioned dataset as specified in the DIME INI being used. The records making up the PDF documents will be sized according to the LRECL although we recommend a minimum length of 256. DOC1DIME is submitted to the system using standard JCL.

EXEC card syntax:

```
EXEC PGM=DOC1DIME,PARM=('INI=DD:IniDD [,symbol=value,symbol=value,...]')
```

Parameters:

IniDD	is the DD label in the JCL that indicates the DIME INI to be used.
symbol=value	indicates a value to be used to replace a symbol in the DIME file. This is optional and may be repeated as many times as required, each instance separated by a comma.

Examples:

Specifying initialization file:

```
//DOC1DIME EXEC PGM=DOC1DIME,PARM='INI=DD:DOC1INIT'
```

Specifying initialization file and symbols:

```
//DOC1DIME EXEC PGM=DOC1DIME,PARM=('INI=DD:DOC1INIT,VAL1=EARLY')
```

Sample JCL:

```
//Jobname JOB (xxx) ...(Rest of Job Card parms)
//DOC1DIME EXEC PGM=DOC1DIME,
// PARM='/INI=DD:DIMEINI'
//*Load lib for PCE + run-time libs if req’d
//STEPLIB DD DISP=SHR,DSN=PROD.GEN.LOAD
//*Initialization file (DD referenced in EXEC)
//DIMEINI DD DISP=SHR,DSN=PROD.GEN.RUN(A1INI)
//*Generate messages file (DD referenced in INI)
//DOC1MSG DD DISP=SHR,DSN=PROD.GEN.MSG(MESSAGES)
//*Input PDF file (DD referenced in INI)
//PDFIN DD DISP=SHR,DSN=PROD.GEN.HIPS(JOB10))
//*
```

10 - Working with HTML

HTML output produced by Generate includes some features of the DHTML (dynamic HTML) specification which is a development of the original HTML tags incorporating cascading style sheets, layers, dynamic fonts and other programmable options.

> **Note:** that not all internet browsers can display DHTML and standards can vary between those that can. Generally speaking the DHTML produced by Generate should be compatible with any browser supporting cascading style sheets such as the 4.0 or later versions of Internet Explorer and Netscape Communicator.

As with all other datastreams produced by Generate HTML output is produced by as a single stream containing all the documents generated by the application. The individual HTML pages cannot be browsed without subsequent manipulation. The HTML output is actually an XML structure known as a PAK file which provides a structured container for the HTML pages and their associated resources.

If you intend to pass the HTML output to Vault this format is automatically interpreted and you need only pass the entire stream to the download directory for processing. For other applications you will need to use an extraction utility to separate the pages and objects within the PAK file and make them available to your presentation system.

The DOC1EDU program is provided as a simple file-based example of an extraction utility for a Generate HTML PAK file. See **“EDU” on page 243** for more information.

In this section

Deployment considerations

Extract and deployment systems need to do at least the following:

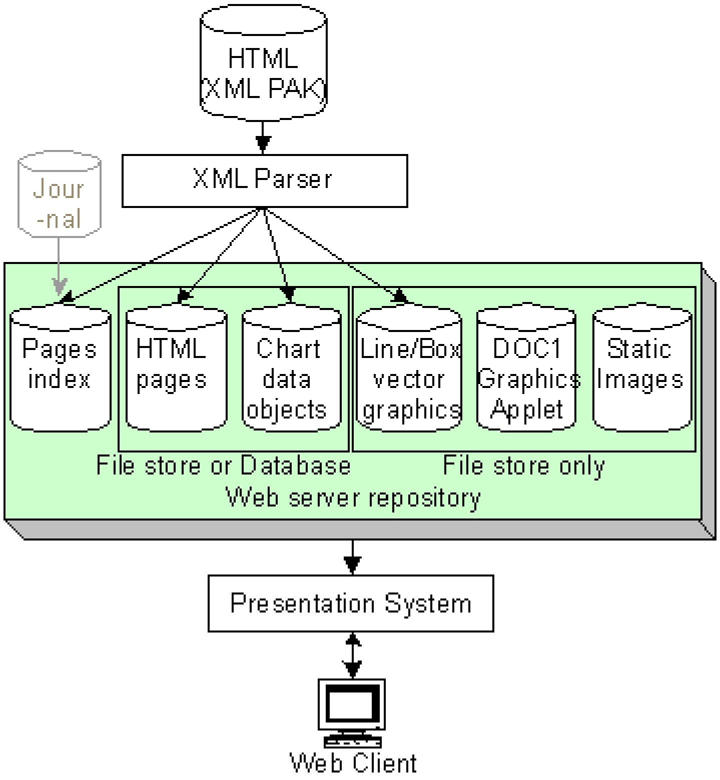
- Extract the HTML pages from the PAK file to separate files on your database/file store and log the relevant identifiers so they can be indexed by your web server system.
- Extract any native chart data objects as separate files to your database/file store. This must match the location specified in the **Chart data location** output device setting used when publishing the job from Designer.
- Extract the line and box vector graphics to your file store at the location specified by the Resource location output device setting.
- Make any static image files referenced by the application available to the web server at the location defined in the Image location output device setting.
- Access the relevant HTML pages via their logged identifiers when a display request is made.

Note that where native charts are used the DOC1 Graphics Applet will also need to be stored in the resource location defined by the Applet location option so that it can be downloaded on demand.

Refer to the Output, media and resources section of the Designer User's Guide for detailed information regarding HTML output device settings.

DHTML deployment model

For each HTML page created the deployment system will need to extract three different object types from the PAK file as indicated in the diagram.



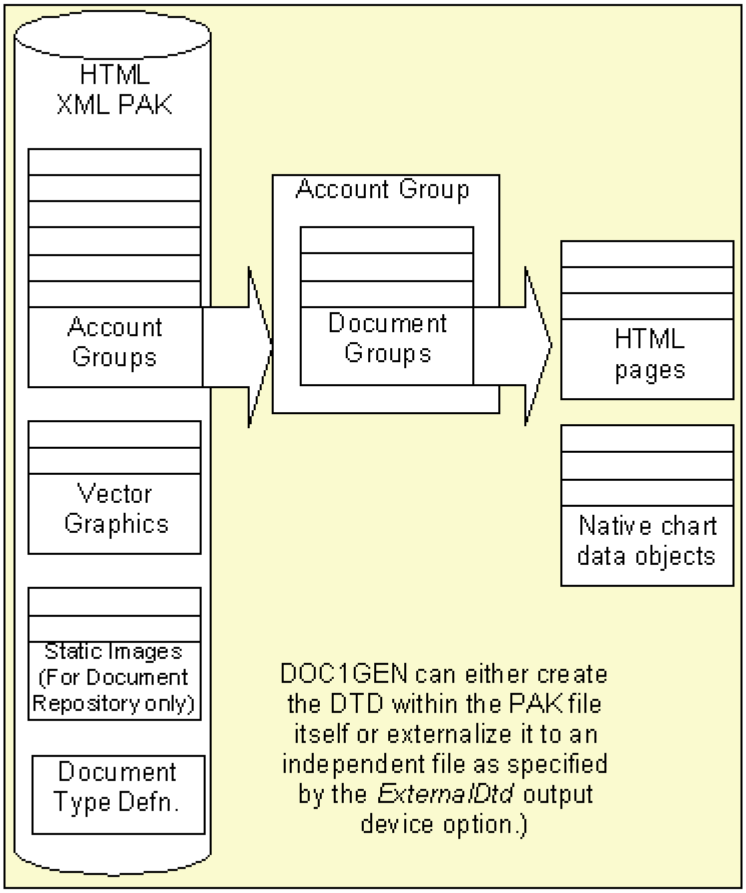
The HTML pages themselves can be extracted to the location required by your presentation system. The two types of dynamically generated resources also created within the PAK must be extracted to the locations indicated by the relevant output device options.

Static images and the DOC1 Graphics Applet used to present native charts are assumed by the HTML to be already present at the location indicated by the Resource location output device option.

The extraction routine will also need to build an index to the HTML files so they can be located by your presentation system. You can use the group, document and page ID attributes within the XML elements to do this, or use a journal file to supply additional references.

PAK file structure

The HTML PAK file created by Generate is an XML construct providing a container for HTML pages and associated resources along with a Data Type Definition (DTD) that describes the file structure.



DOC1GEN can either create the DTD within the PAK file itself or externalize it to an independent file as specified by the *ExternalDtd* output device option.)

The use of XML and the DTD means that the PAK file can easily be parsed using a range of publicly available utilities.

HTML pages within the PAK file are grouped at two levels:

- The HTML pages produced by a single publication data set are stored within a document group.
- One or more document groups are stored within an ‘account’ group. All sequential documents sharing the same XML attribute groupID will be placed in the account same group. The groupID attribute can be specified as the keyword parameter of the document attributes object when designing the application in the Designer. If this is omitted a default groupID is assigned by DOC1GEN and each document will be stored in an individual account group.

All elements of the PAK file have unique identifier attributes which you can use to cross-reference and further customize your system as required.

HTML PAK file elements and attributes

dhtmlpak

The overall structure of HTML PAK file. Will always contain one or more group elements.

jobdata

This will describe the environment in which the PAK was generated.

Attributes:

date – system date time – system time platform – operating platform for which the resources are intended. One of: Windows, UNIX, z/OS.

group

The ‘account’ level group structure. Will contain zero or more document elements. May also contain other elements which are intended for future expansion: cpage, fpage, and docmap. All sequential documents that share the same groupID attribute will be contained within the same group. If no groupID is available an account level group will be created for each individual document. **Attributes:**

groupUID – a unique identifier for the group generated dynamically be DOC1GEN. This can be used to access the group by the extract mechanism.

groupID – the group identifier. If the publication included a document attributes object the keyword provided as part of this will be used as the groupID. If not a unique identifier will automatically be assigned by DOC1GEN.

document

The document level group structure. Will contain one or more dhtmlpage elements. Can also contain zero or more img and jdata elements.

Attributes:

docUID – a unique identifier for the document generated dynamically be DOC1GEN. This can be used to access the document by the extract mechanism.

docID – a reference indicating the sequence of the document within the group. There may be other attributes relating to specific cross application requirements.

dhtmlpage

Contains individual HTML pages.

Attribute: pageID – a reference indicating the sequence of the page within the document.

graphic

Contains individual line and box images as generated by DOC1GEN. Attributes: imUID – a unique reference for the image url – the URL reference (built using ResourceURL INI setting) datalen – contains the number of bytes making up the img element (excluding carriage returns)

jdata

Contains the data to be used as parameters when constructing a native chart object with the DOC1 Graphics Applet.

Attributes:

jdUID – a unique reference for the object datalen – contains the number of bytes making up the jdata element (excluding carriage returns)

TOC	Appears at the end of each document when document section bookmarks have been defined for the document. When used for Vault it is used to build a page navigator for documents. Otherwise it can be used to generate a bookmark structure for those intending to build their own web site map. Consists of Bmark tags which themselves contain these attributes: name – is a reference name pageid – which page within doc lref – position on the page
cpage, fpage, docmap	Future expansion only. No relevant content at this time.

Graphics handling

Unlike other datastreams supported by Generate, HTML has no concept of vector drawing and special methods need to be employed to deliver such elements to the client system.

Lines and boxes

Native lines and boxes required by the application are created as independent vector graphic files (of type GIF) and included in the XML PAK as independent objects. A single object is not repeated and can be referenced by any number of HTML pages.

The HTML pages will reference these files at the URL specified as the Resource Location output device setting.

Static images

In Generate terms these are the referenced elements that can be included in documents.

Images used in your design are converted to png format. By default, these images are identified by their original base name together with an automatically generated identifier. Use the resource mapping facility if you wish to override the generated name. Refer to the resource maps section of the Designer User’s Guide for further information.

As with lines and boxes the HTML pages will reference these files at the URL specified as the Resource Location output device setting.

Designer native chart feature

Native chart graphics are rendered dynamically on the client system by a Java applet supplied with product distribution material. The parameter data is passed to the applet either as meta-data within the HTML page itself or, where large amounts of data is involved, via a chart data object that is generated within the XML PAK by DOC1GEN. The decision to produce an independent object is based on the **Chart threshold** output device setting which specifies an upper size limit under which inline data will be used. The chart data objects need to be extracted to database or file store locations that have been anticipated in the **Locate chart data by** and **Chart data location** output device settings.

Locate chart data by can be either: URL – indicating a URL path Query – indicating a query string (for database look-up etc.)

The examples below show possible Chart data location settings:

Locate chart data by set to URL:

```
http://gen/resources/&jdUID.dat
```

Uses fully qualified path and jdUID as file name

```
&jdUID.dat
```

Uses current path/default location

```
http://gen/&docUID/&pageUID/&jdUID.dat
```

Use several unique IDs to provide full path

Locate chart data by set to Query:

```
http://gen/scripts/getcd.cgi?&jdUID
```

Calls a cgi script in the defined location using the jdUID attribute as the sole parameter.

```
DataURL = getcd.asp?/&pakUID&docUID&jdUID
```

Calls an asp script in the current path/default location using several unique IDs as parameters.

EDU

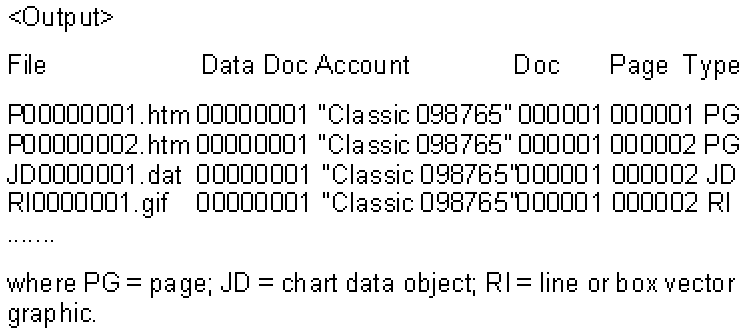
The DOC1EDU (Extract and Deployment utility) program is provides as a simple file-based example of an extraction utility for an HTML PAK file. Note that this is recommended as a tool to assist with your application development; production systems will typically require a custom deployment utility.

EDU interrogates a PAK file and extracts the individual HTML pages and the resources required to present them to the file locations you specify. EDU is controlled by an initialization file (INI) that identifies the input file and locations for output files.

The EDU (Extract and Deployment Utility) can extract the HTML pages and associated resources from the PAK file and store them as individual files. These can then be viewed by an Internet browser.

EDU index

EDU optionally produces an index to the HTML pages and associated resources that have been extracted during execution. This contains some informational sections followed by a listing of each page or resource as shown in this fragment:



```
<Output>

File              Data Doc Account        Doc     Page  Type

P00000001.htm 00000001 "Classic 098765" 000001 000001 PG
P00000002.htm 00000001 "Classic 098765" 000001 000002 PG
JD0000001.dat  00000001 "Classic 098765"000001 000002 JD
RI0000001.gif   00000001 "Classic 098765"000001 000002 RI
.......

where PG = page; JD = chart data object; RI = line or box vector
graphic.
```

EDU Initialzation file format

The Initialization file specifies the environment for a particular execution of DOC1EDU.

Parameters are normally coded as literal values but can be specified dynamically if required by defining them wholly or partly as symbols. Values can be assigned to such symbols when starting the DOC1EDU program. Within the INI file symbols are referenced by coding the symbol name to be used within percentage signs. For example: `Input=%FName%`

Syntax:

```
<Files>

Input=Location

PageFileMask=Location%1

ResourceFileMask=Location%1

IndexFile=Location

DtdFile=Location

Messages=Location
```

Data types:

Location

is a reference to a file location in the format required by the host operating system. For file mask parameters the wildcard (%1) should be included to indicate the position of the actual filenames in the path structure. Examples: Windows – c:\doc1host\dhtmlout\%1 UNIX – /doc1host/dhtmlout/%1 z/OS – DD:DHTMLOUT(%1)

The Preface of this manual contains guidelines for the expected naming conventions for all supported platforms.

Keyworda and parameters:

Input

Location is the path/filename of a HTML PAK file (as produced by DOC1GEN) which is to be input to EDU.

PageFileMask

Location is the path where the individual HTML pages will be output. The %1 wildcard will be substituted by a unique reference by EDU.

ResourceFileMask	Location is the path where resource file associated with the HTML pages will be output. These may be either: vector graphic files (of type GIF) which are created for each line or box shape required by the HTML pages native chart data files containing parameters to be passed to the DOC1 graphics applet so that charts can be rendered The %1 wildcard will be substituted by a unique reference in both cases. If ResourceFileMask is omitted the Location specified for PageFileMask will be used
IndexFile	Location is the path/filename which will receive the index produced by EDU. Size will vary according to number of pages and resources produced by the application. Allow approximately 50 bytes per page or resource.
DtdFile	Location is the path/filename which will receive the XML Data Type Definition (DTD) that describes the PAK input file. This keyword is optional – if omitted the DTD is not extracted.
Messages	Location is the path/filename of the EMFE error messages file (`messages` on z/OS or `messages.dat` on Windows, Unix etc). This keyword is mandatory.

Example:

Under Windows, UNIX, etc.:

```
<Files>
Input=html.pak
PageFileMask=c:\doc1host\dhtmlout\%1
ResourceFileMask=c:\doc1host\dhtmlout\resource%1
IndexFile=c:\doc1host\dhtmlout\pakindex.dat
Messages=c:\doc1host\dhtmlout\messages.dat
```

Under z/OS:

```
<Files>
Input=DD:DHTMLPAK
PageFileMask=DD:DHTMLOUT(%1)
ResourceFileMask=DD:DHTMLOUT(%1)
IndexFile=DD:DHTMLINX
Messages=DD:MESSAGES
```

Running DOC1EDU

Running DOC1EDU under UNIX & Windows

Purpose: Extracts HTML pages and associated resources from a Generate HTML PAK file.

Preparation: DOC1EDU is run from the command line of an appropriate operating system window.

Syntax:

doc1edu ini=Ini

Parameters:

INI Ini is the path/filename of the EDU Initialization File to control this execution of DOC1EDU.

Example:

```
doc1edu ini=\doc1host\run\edu.ini
```

Running DOC1EDU under z/OS

Purpose: Extracts HTML pages and associated resources from a Generate HTML PAK file.

Preparation: It is anticipated that the HTML and resource locations will be partitioned datasets. The directory block allocation must allow for all expected files.

Note that the DD labels are examples only – the actual label required are specified in the INI file being used.

JCL:

```
///Jobname JOB (Rest of Job Card parms)
//DOC1EDU EXEC PGM=DOC1EDU,PARM='INI=DD:IniDD'
//STEPLIB DD DISP=SHR,DSN=EDU Load Library
//*        Other run time libraries as required (if any)
//IniDD DD DISP=SHR,DSN=Dataset containing EDU INI file
//HTMPAGES DD DISP=SHR,DSN=Dataset to receive HTML pages
//HTMRES DD DISP=SHR,DSN=Dataset to receive related resources
//HTMINDEX DD DISP=SHR,DSN=Dataset to receive index of pages/resources
//HTMDTD DD DISP=SHR,DSN=Dataset to receive XML DTD for the input file
//*
```

Parameters:	
IniDD	identifies the DD name used to define the dataset containing the EDU Initialization File to control this execution of DOC1EDU.
DD References:	All references other than that for the Initialization file itself are coded in the EDU Initialization file.

11 - User exits

A user exit allows Generate to initiate a user–defined program and for the program to return data to the Generate production job if appropriate. Theoretically, a user exit can perform any function but in practice they are limited by the interfaces provided by Generate and the return data it expects.

In this section

Compatibility

The user exit mechanism discussed in this section is only available with the Series 5 DOC1GEN production engine. If you need to use user exits with DOC1PCE these will continue to use the mechanism provided with the DOC1 Suite 4 environment. You should refer to any version of the Programmer’s Guide provided with DOC1 Suite 4 for information about how such exits are coded and implemented. Contact your Precisely support for further information.

Types of user exits

At present four types of user exit are supported:

- A ***File*** user exit allows you to replace the I/O function of a specific file used by Generate. You may want to use such an exit to pre-process input sources before delivering blocks of data to Generate or to direct blocks of output data to a third party handling mechanism such as an API or messaging system
- A ***Data Input*** user exit allows you to completely replace the Generate mechanism for accessing input data. Unlike a file exit, the user program takes responsibility for creating the appropriate data structure expected by an entire publication before passing it to Generate. You may want to use this type of exit if you need to merge input sources dynamically or access databases during the actual Generate production process.
- A ***Key Map*** user exit allows you to access custom sources to insert images where an external keyed object has been used in a publication design. For more information see “Defining external keyed images” on page 215.
- A ***Lookup Table*** user exit allows you to access custom sources to substitute strings when lookup table functions are used in a publication design. Rather than use fixed tables the exit allows the user program to respond dynamically to each call made to a lookup table function.

Where used, a user exit must completely replace the function it relates. For instance, if you code a File exit it will be required to handle all calls that Generate may make in relation to the files including open, close, read, write and so on.

Preparing Generate for User Exits

User exits are notified to Generate by coding the keyword USEREXIT(name) for the appropriate setting in the Override Production Settings file (OPS) as used by Generate. For example:

```
Output1=USEREXIT(SendToArchive)
```

Note: For:

- Data Input exits: use this format with the `DataInput` keyword.
- File exits: use with any input or output keyword.
- Key Map exits: use with any entry in the `<KeyMap>` section.
- Lookup Table exits: use with any entry in the `<LookupTable>` section.

The name used ('SendToArchive' in the example above) must match the external name specified in the appropriate `DuxRegister`... function within the user program itself.

A `<UserExit>` section is required to identify the object modules containing the referenced user programs.

If required, you can pass parameters to the user program either by accessing symbols defined in the OPS file or by extending the parameters in the `USEREXIT` keyword. The `DuxGetOpsSymbolValue` and `DuxGetInvokeParameter` API functions allow you to retrieve these values within the user program.

The format of these commands is demonstrated in the OPS example below.

Note: refer to the Designer User's Guide for general information about creating and using an OPS file.

Sample OPS settings for user exits

```
;replace input data with a user exit
<Input>
DataInput=USEREXIT(BWDI)

;The user exit will provide keyed images
<KeyMap>
Map1=USEREXIT(AKM)

;replace a lookup table with a user exit
;the user program expects 2 parameters
<LookupTable>
Table1=USEREXIT(BWLT,15,2004)

;this declares the user program modules
```

```
<UserExit>
1=DOC1IN.DLL
2=DOC1KMAP.DLL
3=DOC1LTAB.DLL
```

Creating the user program

The user program interfaces with Generate using the DUX API (User Exit) which is an external function library supplied with the Generate distribution material. This object module must be included in the link dependencies when the user program is built. A header file declaring the available functions is also supplied and this must be included in the source code for the user program.

The API is written in standard C and this is also the language expected in the user program. C++ programs can also access the interface but note that no C++ classes are provided.

Note:

- If the program calls modules in other languages it is the users responsibility to ensure that all inter-language linking issues are properly resolved.
- Under z/OS you may only initiate one user exit function for each load module.
- Under Windows all user exit functions called by a single Generate program should be contained within a single DLL to prevent memory fragmentation.

Program structure

All user exit programs must consist of at least:

A DOC1REG function (the name is fixed) which registers the user program with Generate. Within DOC1REG you will need to include a call to the relevant DuxRegister… API function which defines the type of user exit to Generate, supplies a name by which the exit is identified externally and names a further function that will carry out the actual user exit activity.

The user exit function which must be coded to respond to each of the request types that may be made by Generate. The list of possible requests vary according to the type of exit.

Note: See “Code Samples” on page 284 for examples of this program structure as it relates to specific types of user exit.

Multi-threading & user exit handles

User exits support multi-threading – i.e. the same user function can be called multiple times within the same Generate process. This in turn allows the exit to be used in scenarios where Generate needs to execute multiple instances of its engine. To allow multi-threading, each time a user exit is called from Generate it is automatically passed a handle (hdux) that uniquely identifies that particular instance of the exit. This handle is then used as a parameter when the user exit program makes any calls to API functions.

The user program itself should not use any techniques that would prevent multi-threading. In particular the use of static data should be avoided wherever possible. You may use the pvUser structure to pass global data between user exit events if required. You should also check the relevant information for your production platform and compiler for advice on programming for a multi-threaded environment.

Preparing input data

Where applicable, the user program must deliver input data in the structure and sequence that Generate expects; i.e. it must conform to the data format that was assigned to the application in the Designer.

Compilation & run-time requirements

The DUX API object module is supplied separately for each supported platform with product distribution material. Your user program must be linked to the appropriate object module.

Windows

The user program must be exported as a DLL. At run time it must be in the same directory as DOC1GEN or a path identified in a system environment setting.

UNIX

User programs must be generated as UNIX shared objects or equivalent. On most supported UNIX platforms shared object status is achieved by using an appropriate link option:

Unix platform	Option
Sun	`-G -K pic -lelf`
Under AIX the module must be exported. For example:	
AIX	`-bE:name.exp -bM:SRE`

z/OS

The user program and DUX API object modules must be included in the load library concatenation in the STEPLIB statement of the appropriate Generate start-up JCL or in a library in the active link list.

Programming guidelines & function overview

This section summarizes the way the user exit program should be coded and the DUX API functions that must be used. For full details of each API function see “DUX API function library” on page 259. You may also want to refer to “Code Samples” on page 285 for practical examples of the required program structure for each type of exit.

A DOC1REG function must be included in every program to be used as a user exit. It must use the following prototype:

```
extern "C" __declspec(dllexport) DUXRC DOC1REG (HDUX hdux)
```

It must include at least a ***DuxRegister***… function which defines the type of user exit to Generate, supplies a name by which the exit is identified externally and names a further function that will carry out the actual user exit activity. You may also want to code DuxRegister…Event calls to identify other user functions to be called in response to the start and end of the Generate job.

The user exit function name must match that specified in the registration function. Its parameters will vary depending on the type of exit being used.

Interfaces & job control

To access symbols defined in the OPS file or the parameters in the `USEREXIT` keyword use the `DuxGetOpsSymbolValue` and `DuxGetInvokeParameter` functions. `DuxGetInvokeParameterCount` returns the number of available parameters.

The user program may output operator messages using the standard Generate reporting mechanism. Use the `DuxInformUser` and `DuxWarnUser` to output messages with an information or warning level respectively.

> **Note:** information and warning messages may be suppressed by settings assigned to the production job.

In some circumstances the user exit program may encounter an error condition requiring processing to be halted. This situation can be handled using the `DuxStopJob` function which allows you to issue an abort level message and makes a request to Generate to stop the job.

> **Note:** Generate may still need to call the user program again after `DuxStopJob` has been called – for example, a file user exit may be requested to close its virtual file – and so should be written to cater for such scenarios.

> **Note:** If `DuxStopJob` is called when Generate is configured in server mode just the job is stopped and not the process. The user program will be called again when the next job is available for processing.

Return codes

The user exit function will be expected to issue a return code. Additionally, return codes are issued by many DUX API functions for query be the user program if required. All valid return code values are defined as constants in the user exit header file as supplied.

Working with data types

To ensure compatibility with the internal data types required by Generate you must first store strings and numbers in data objects before passing them to DUX API functions.

String objects can contain either 8-bit or 16-bit characters. You can allocate a string object using `DuxStrAlloc` or both allocate and assign data using the appropriate `DuxStrAllocWithString…` function. You can update the contents of a string object using `DuxStrSet…` functions and retrieve a string from an object using `DuxStrGet…` functions. Use `DuxStrFree` to free the memory assigned to a string object if required.

Use number objects to store numeric or currency values. Note that when updating number objects you will need to specify the whole and fractional parts as separate values. Allocate a number object using `DuxNumAlloc` and assign a value using `DuxNumStore`. Use `DuxNumFree` to free the memory assigned to a string object if required.

Code page management

Generate interprets strings using a code page which will be assumed from the production platform. You may need to instruct Generate to use a different code page particularly when dealing with text based on multiple languages. The range of supported code pages are defined in the user exit header file as supplied.

Use the `DuxGetCodePage` function to return the code page currently being used as the default. You can set a new default code page using `DuxSetCodePage`. You can also set the code page to be used with specific string objects using `DuxStrCodePage`.

Coding a File or Data input user exit

The DOC1REG function must include a call to the relevant registration API function:

- `DuxRegisterFileInputExit` for a file exit to handle input data;
- `DuxRegisterFileOutputExit` for a file exit to handle output data;
- `DuxRegisterDataInputExit` for a data input exit.

These functions specify the name of the actual user exit function. The user exit function itself should be declared using the following prototype:

```
DUXRC name (HDUX hdux, DUXDATAACTION action, void *pvUser)
```

Where:

- `name` is the name of the function as identified as part of the DuxRegister… call in the DOC1REG function;
- `hdux` receives the unique handle to the user exit session from Generate; action will receive requests from Generate in the form of constants defined in the user exit header file;
- `pvUser` may be used to pass private data between user exit events if required.

Defining file types

For file exits you will need to be aware of the expected access method for the file and the type of data to be read/written.

Generate supports two access methods: record or stream. Record based files are delimited either using native record structures under z/OS or with carriage return or carriage return/line feed bytes on other platforms. Reading or writing to stream based files simply deals with un-delimited blocks of data and is typically used with binary data.

This information is passed to Generate using the `flFlags` parameter of the relevant `DuxRegisterFile`…Exit function.

Handling requests

For these types of exit Generate will issue a range of requests – or actions – to the user program. The user exit function must be coded to respond to each of the following actions and include the appropriate API functions.

File exits must respond to the following actions:

`DUXFA_Open` requests that the relevant file be opened.

`DUXFA_Read` requests data from the file (input exits only). The user program will normally use the `DuxSetFile`… functions to access the Generate file buffer and return a record or block of data (according to how the exit was defined in `DuxRegisterDataInputExit`).

`DUXFA_Write` requests that data be written to the file (output exits only). The user program will normally use the DuxGetFile… functions to access the Generate file buffer and get access to each record or block of available data (according to how the exit was defined in `DuxRegisterDataInputExit`).

> **Note:** Generate automatically grows the I/O buffer to accommodate the data when using the functions discussed above. Other API functions allow you to specifically control the buffer size but note that incorrect control of the buffer will cause Generate to fail.

`DUXFA_Tell` requests the user program to provide the current file offset. Use `DuxGetFileOffset` to return the current position.

`DUXFA_Seek` requests the user program to move the file pointer to a particular position. This is based on a specified offset from the last position indicated by `DuxGetFileOffset`. Use `DuxSetFileOffset` to move to the required position in the file.

DUXFA_IsEndOfFile requests the user program to check the availability of data. The user program should return DUXRC_DataAvailable or DUXRC_EndOfData as appropriate. DUXFA_Close requests the user program to close the file.

Data input exits must respond to the following actions:

DUXDA_VirtualOpen requests the data source to be opened in preparation for input. Typically the user program will establish a database connection, open files or similar.

DUXDA_ReadPublication requests the user program to assemble the data structure required by the next publication to be processed. You will need to use the API functions to build your data into the structure defined by the data format being used. You can establish the required structure using the Data Format Editor in the Designer.

Typically you will need to define an instance of a each expected input record using DuxAddRecord and then add field values to the records using the relevant DuxAdd…Field function.

All records other than a start of publication record must have the appropriate parent record defined in the DuxAddRecord call. For start of publication records the parent parameter should be NULL.

DUXDA_IsEndOfData queries the availability of further data from the user program. The user exit function must issue a return code of DUXRC_DataAvailable or DUXRC_EndOfData in response.

DUXDA_VirtualClose requests the data source to be closed, usually when there is no more data available, or when an error has occurred.

Coding a Key Map user exit

The DOC1REG function must include a call to the DuxRegisterKeyMapExit API function. Caching is used to improve performance and is normally handled automatically by Generate. If you need to use a custom caching you should set the appropriate flag in the registration function. The user exit function itself should use DuxStrGetChars to access the key being queried and should use DuxStrSetString and DuxNumFromINT to return the image and its properties.

> **Note:** See the Key Map User Tutorial document on the website for detailed technical information.

Coding a Lookup Table user exit

The DOC1REG function must include a call to the DuxRegisterLookupTableExit API function. Caching is used to improve performance and is normally handled automatically by Generate. If you need to use a custom caching you should set the appropriate flag in the registration function. The user exit function itself should use DuxStrGetChars to access the lookup key being queried and DuxStrSetString to return the substitute value.

Code samples

This section contains samples of the various user exit types supported by Generate. It should be noted that these samples are for your guidance as the design of the user exit is completely at your discretion.

The sample code extracts provided in this section do not show the entire program required to implement the particular user exit functionality, as it focusses on the key program structures required to implement the chosen user exit type.

Data input user exit example

```
extern "C" __declspec(dllexport) DUXRC DOC1REG (HDUX hdux)
               {
               PDUXSTRING pstr = DuxStrAlloc (hdux);
               //
               // Register our data input user exit with the name 'BWDI'.
               //
               DuxStrSetString (hdux, pstr, "BWDI");
               DuxRegisterDataInputExit (hdux, pstr, MyDataInputExit, mydata);
               //
               // Register when Generate terminates so I know when to free 'mydata'.
               //
               DuxRegisterTerminateEvent (hdux, FreeMyData, mydata);
               //
               // Let the user know that we've been loaded.
               //
               DuxStrSetString (hdux, pstr, "Boardwalk Database Interface ready for use.");
               DuxInformUser (hdux, pstr);
               DuxStrFree (hdux, pstr);
               return DUXRC_OK;
               }
               extern "C" DUXRC MyDataInputExit (HDUX hdux, DUXDATAACTION action, void *pvUser)
               {
               DUXRC rc = DUXRC_OK;
               switch (action)
               {
               //
               // Generate is requesting a data source to be opened in preparation for input.
               //
               case DUXDA_VirtualOpen:
               {
               PMYDATA pmydata = (PMYDATA)pvUser.
               // If database opens fails
               if (pmydata->precdata->Open () == 0)
               {
               pmydata->fEof = TRUE; // So we don't try to fetch data.
               rc = DUXRC_Failed; // Have Generate abend.
               }
               else
               {
               PDUXSTRING pstr = DuxStrAlloc (hdux);
               // Just let user know that we've connected to database.
               DuxStrSetString (hdux, pstr, "Boardwalk Database connection success.");
               DuxInformUser (hdux, pstr);
               DuxStrFree (hdux, pstr);
               pmydata->fEof = FALSE;
               }
               }
               break;
               //
               // Generate is requesting a data source be closed as it is no longer required.
               //
               case DUXDA_VirtualClose:
               {
               PDUXSTRING pstr = DuxStrAlloc (hdux);
               pmydata->precdata->Close();
               DuxStrSetString (hdux, pstr, "Boardwalk database Closed");
               DuxInformUser (hdux, pstr);
               DuxStrFree (hdux, pstr);
               }
               break;
               //
               // Generate is asking if there is any more publication's worth of data.
               //
               case DUXDA_IsEndOfData:
               if (pmydata->fEof)
               rc = DUXRC_EndOfData;
               else
               rc = DUXRC_DataAvailable;
               break;
               // Generate is asking for a publications worth of data.
               //
               case DUXDA_ReadPublication:
               // Create the publication record.
               //
               DuxStrSetString (hdux, pstrKey, "1000");
               precPub = DuxAddRecord (hdux, NULL, pstrKey);
               // Add the account details.
               //
               DuxStrSetString (hdux, pstrName, "Acct No");
               DuxStrSetString (hdux, pstrValue, pmydata->precdata->m_AcctNum);
               DuxAddStringField (hdux, precPub, pstrName, pstrValue);
               DuxStrSetString (hdux, pstrName, "Cust Name");
               DuxStrSetString (hdux, pstrValue, pmydata->precdata->m_CustName);
               DuxAddStringField (hdux, precPub, pstrName, pstrValue);
               DuxStrSetString (hdux, pstrName, "Addr 1");
               DuxStrSetString (hdux, pstrValue, pmydata->precdata->m_Addr1);
               DuxAddStringField (hdux, precPub, pstrName, pstrValue);
               // Create the summary details record.
               //
               DuxStrSetString (hdux, pstrKey, "1500");
               prec = DuxAddRecord (hdux, precPub, pstrKey);
               // Add the details.
               //
               DuxStrSetString (hdux, pstrName, "Current Total");
               DuxStrSetString (hdux, pstrValue, pmydata->precdata->m_CurrentTotal);
               DuxAddStringField (hdux, prec, pstrName, pstrValue);
               DuxStrSetString (hdux, pstrName, "Amount Last Paid");
               DuxStrSetString (hdux, pstrValue, pmydata->precdata->m_AmountLastPaid);
               DuxAddStringField (hdux, prec, pstrName, pstrValue);
               break;
               default:
               rc=NoParm;
               break;
               }}
               return rc;
               }
```

File user exit example

```
extern "C" __declspec(dllexport) DUXRC DOC1REG (HDUX hdux)
{
PDUXSTRING pstr = DuxStrAlloc (hdux);
DUXINT32 nFlags;
//
// Set the flags indicating the type of IO we support. This user exit just
// supports reading of record mode interface.
//
//
nFlags = DUXFF_Read | DUXFF_RecordMode | DUXFF_Binary;
//
// Register our File Input User Exit with the name 'BWFI'. It doesn't need to
// be the same name as the DLL. It just happens to be.
//
DuxStrSetString (hdux, pstr, "BWFI");
DuxRegisterFileInputExit (hdux, pstr, nFLags, MyFileInputExit, mydata);
//
// Register when Generate terminates so I know when to free 'mydata'.
//
DuxRegisterTerminateEvent (hdux, FreeMyData, mydata);
//
// Let the user know that we've been loaded.
//
DuxStrSetString (hdux, pstr, "Boardwalk File Input Interface ready for use.");
DuxInformUser (hdux, pstr);
DuxStrFree (hdux, pstr);
return DUXRC_OK;
}
extern "C" DUXRC MyFileInputExit (HDUX hdux, DUXFILEACTION action, void *pvUser)
{
DUXRC rc = DUXRC_OK;
switch (action)
//
// Generate is requesting the file to be opened.
//
case DUXFA_Open:
{
PMYDATA pmydata = (PMYDATA)pvUser.
//
// Ensure there are two parameters specified when we are invoked.
// If not, tell the user what they need to do.
//
if (DuxGetInvokeParameterCount (hdux) < 2)
{
...
rc = DUXRC_Failed;
}
//
// Otherwise we’ll open up the two files specified and raise an error
// if we can’t find them.
//
else
{
PDUXSTRING pstrFile1 = DxGetInvokeParameter (hdux, 1);
PDUXSTRING pstrFile2 = DxGetInvokeParameter (hdux, 2);
...
}
}
break;
//
// Generate is requesting to read a record from the file.
//
case DUXDA_Read:
{
FILE *pfToUse = null;
// Figure out which file to use for the read.
if (!feof (pmydata->pfFile1))
pfToUse = pmydata->pfFile1;
else if (!feof (pmydata->pfFile2))
pfToUse = pmydata->pfFile2;
// If we have a file with data in it...
if (pfToUse != NULL)
{
char *pchEol = NULL;
// Read data from file.
fgets (pmydata->szBuffer, sizeof (pmydata->szBuffer)-1, pfToUse);
// Remove the end of line character. Generate doesn’t want it and will
// treat it as data.
puchEol = strchr (pmydata->szBuffer, '\n');
if (puchEol != NULL)
 *puchEol = '\0';
// Give Generate the record data.
DuxSetFileData (hdux, pmydata->szBuffer, strlen (pmydata->szBuffer));
}
// Otherwise we don’t have anymore data.
else
rc = DUXRC_EndOfData;
break;
//
// Generate is asking if there is any more data to read.
//
case DUXDA_IsEndOfData:
if (feof (pmydata->pfFile1) && feof (pmydata->pfFile2))
rc = DUXRC_EndOfData;
else
rc = DUXRC_DataAvailable;
break;
//
// Generate is requesting that the file be closed.
//
case DUXDA_VirtualClose:
{
PDUXSTRING pstr = DuxStrAlloc (hdux);
fclose (pmydata->pfFile1);
fclose (pmydata->pfFile2);
DuxStrSetString (hdux, pstr, "Boardwalk File Input Exit Closed");
DuxInformUser (hdux, pstr);
DuxStrFree (hdux, pstr);
}
break;
*/
}
return rc;
}
```

Keymap user exit example

```
extern "C" __declspec(dllexport) DUXRC DOC1REG (HDUX hdux)
{
PDUXSTRING pstr = DuxStrAlloc (hdux);
KUXDATA* pData = KUxDataNew ();
//
// Register our Keymap User Exit with the name 'AKM'. It doesn't need to
// be the same name as the DLL. This is the name by which it is referred to
// in the hip file - USEREXIT(AKM)
//
DuxStrSetString (hdux, pstr, "AKM");
DuxRegisterKeyMapExit (hdux, pstr, DUXLTF_NoCacheResults, KUxPdfDataEntry, pData);
//
// Register when Generate terminates so that 'pData'can be freed.
//
DuxRegisterTerminateEvent (hdux, KUxDataDelete, pData);
//
// Let the user know that we've been loaded.
//
DuxStrSetString (hdux, pstr, "Key Map User Exit Interface is ready for use.");
DuxInformUser (hdux, pstr);
DuxStrFree (hdux, pstr);
return DUXRC_OK;
}
extern "C" DUXRC KUxPsDataEntry (
HDUX hdux, // IN
PDUXSTRING pstrKey, // IN
PDUXNUMBER pnumSeq, // IN/OUT
PDUXSTRING pstrName, // OUT
PDUXSTRING pstrResult, // OUT
PDUXSTRING pstrResType, // OUT
PDUXNUMBER pnumWidth, // OUT
PDUXNUMBER pnumHeight, // OUT
PDUXNUMBER pnumResolution, // OUT
PDUXSTRING pstrDevice, // OUT
void *pvUser)
{
DUXRC rc = DUXRC_OK;
UXDATA* pData = (KUXDATA*)pvUser;
char *pszKey;
int nWidth, nHeight, nResolution, nSeq;
string strName, strValue, strResType, strDevice;
// Initialize if database is not initialised,
if (!pData->fInit)
{
// Any user initialization
// On failure return DUXRC_Failed
pData->fInit = TRUE;
KUxKeyMapNew ();
}
// Get key value.
pszKey = DuxStrGetChars (hdux, pstrKey);
nSeq = DuxNumToINT (hdux, pnumSeq);
//
// Lookup value in database

//
if (KUxKeyMap (pszKey, nSeq, strName, strValue, strResType, nWidth, nHeight,
nResolution, strDevice))
{
// Store the result.
// The name used to reference the image
DuxStrSetString (hdux, pstrName, strName.c_str ());
// The actual image e.g. bitmap drawing instructions
DuxStrSetString (hdux, pstrResult, strValue.c_str ());
// The resource type eg bmp for bitmaps
DuxStrSetString (hdux, pstrResType, strResType.c_str ());
// The device eg ps for postscript
DuxStrSetString (hdux, pstrDevice, strDevice.c_str ());
```

```
// Width of the image
DuxNumFromINT (hdux, pnumWidth, nWidth);
// Height of the image
DuxNumFromINT (hdux, pnumHeight, nHeight);
// Image resolution
DuxNumFromINT (hdux, pnumResolution, nResolution);
// Sequence count in case this DLL serves more than one type of image
DuxNumFromINT (hdux, pnumSeq, nSeq);
return DUXRC_DataAvailable;
}
else
{
return DUXRC_EndOfData;
}
}
//=======================================================================
//
// A user exit handling routine to retrieve a bitmap. It can have almost any name.
// The 'DOC1REG' function at the bottom of the module will tell DOC1GEN
// what it is called.
//
//=======================================================================
extern "C" DUXRC KUxPdfDataEntry (
HDUX hdux, // IN
PDUXSTRING pstrKey, // IN
PDUXNUMBER pnumSeq, // IN/OUT
PDUXSTRING pstrName, // OUT
PDUXSTRING pstrResult, // OUT
PDUXSTRING pstrResType, // OUT
PDUXNUMBER pnumWidth, // OUT
PDUXNUMBER pnumHeight, // OUT
PDUXNUMBER pnumResolution, // OUT
PDUXSTRING pstrDevice, // OUT
void *pvUser)
{
DUXRC rc = DUXRC_OK;
KUXDATA* pData = (KUXDATA*)pvUser;
char *pszKey;
int nWidth, nHeight, nResolution;
string strName, strResType, strDevice;
bytevec bvValue;
DUXINT32 nSeq;
//
// If database is not initialised,
//
if (!pData->fInit)
{
// Any user initialization
// On failure return DUXRC_Failed
pData->fInit = TRUE;
KUxKeyMapNew ();
//
// Get key value.
//
pszKey = DuxStrGetChars (hdux, pstrKey);
nSeq = DuxNumToINT (hdux, pnumSeq);
//
// Get value from database.
//
if (KUxKeyMap (pszKey, nSeq, strName, bvValue, strResType, nWidth, nHeight,
nResolution, strDevice))
{
//
// Populate the parameters that return the image and its properties
//
// The name used to reference the image
DuxStrSetString (hdux, pstrName, strName.c_str ());
// The actual image. This is a binary image but it is returned
// as a string. The receiving end knows it is a binary image and
// will treate it accordingly
PDUXUINT8 pbValue = &bvValue[0];
PCDUXCHAR8 pcValue = (PCDUXCHAR8) pbValue;
DUXUINT32 nValueLen = (DUXUINT32)bvValue.size ();
DuxStrSetChars (hdux, pstrResult, pcValue, nValueLen);
// The resource type
DuxStrSetString (hdux, pstrResType, strResType.c_str ());
// The device type
DuxStrSetString (hdux, pstrDevice, strDevice.c_str ());
// Assign value to image width return value
DuxNumFromINT (hdux, pnumWidth, nWidth);
// Height of the image
DuxNumFromINT (hdux, pnumHeight, nHeight);
// Image resolution
DuxNumFromINT (hdux, pnumResolution, nResolution);
// Sequence count in case this DLL serves out more than one type of image
DuxNumFromINT (hdux, pnumSeq, nSeq);
return DUXRC_DataAvailable;
}
else
{
return DUXRC_EndOfData;
}
}
```

Lookup table user exit example

```
extern "C" __declspec(dllexport) DUXRC DOC1REG (HDUX hdux)
{
PDUXSTRING pstr = DuxStrAlloc (hdux);
//
// Register our Lookup Table User Exit with the name 'BWLT'.
//
DuxStrSetString (hdux, pstr, "BWLT");
DuxRegisterLookupTableExit (hdux, pstr, DUXLTF_CacheResults,
MyLookupTableExit,
mydata);
//
// Register when Generate terminates so I know when to free 'mydata'.
//
DuxRegisterTerminateEvent (hdux, FreeMyData, mydata);
//
// Let the user know that we've been loaded.
//
DuxStrSetString (hdux, pstr, "Boardwalk Lookup Table Interface is ready for
use.");
DuxInformUser (hdux, pstr);
DuxStrFree (hdux, pstr);
return DUXRC_OK;
}
extern "C" DUXRC MyLookupTableExit (HDUX hdux, PDUXSTRING pstrKey, PDUXSTRING
pstrResult, void *pvUser)
{
DUXRC rc = DUXRC_OK;
PMYDATA pmydata = (PMYDATA)pvUser.
char *pszKey;
char szValue[MAX_ValueLength] = "";
//
// Get key value.
//
pszKey = DuxStrGetChars (hdux, pstrKey);
//
// Get value from database.
//
GetValueFromDatabase (pmydata, pszKey, szValue);
//
// Store the result.
//
DuxStrSetString (hdux, pstrResult, szValue);
Return DUXRC_OK;
}
```

User exit API function library

This section lists the functions in the user exit API function library, detailing the function's:

- purpose
- usage
- parameters
- description
- return value(s)

DuxAddCounterField Function

Function:	Adds a field of type Counter to a Data Input record and assigns a value.
Prototype:	`DUXRC DuxAddCounterField (HDUX <hdux>, PDUXPARENT <pParent>, PDUXSTRING <pstrName>, DUXUINT32 <nVal>)`
Parameters:	`hdux` handle to the user exit service. `pParent` pointer to parent record. `pstrName` string; name of the counter field to be added. `nVal` number object.
Description	This function is used only for Data Input user exits. A field defined as type Counter (integer) is appended to the parent record (as created by the previous call to `DuxAddRecord`) and is assigned the value in the number object.
Returns	`DUXRC_InvalidName`: field name is invalid for parent. `DUXRC_InvalidType`: field's definition is not compatible with type counter.

DuxAddDateField Function

Function:	Adds a field of type `Date` to a `Data Input` record and assigns a value.
Prototype:	`DUXRC DuxAddDateField (HDUX <hdux>, PDUXPARENT <pParent>, PDUXSTRING <pstrName>, UINT <nDay>, UINT <nMonth>, UINT <nYear>, DUXCALENDER <cal>)`
Parameters:	`hdux` handle to the user exit service. `pParent` pointer to parent record. `pstrName` string; name of the counter field to be added. `nDay` number; day portion of date field to be added. `nMonth` number; month portion of date field to be added. `nYear` number; Year portion of date field of date field to be added.
Description	This function is used only for Data Input user exits. A field defined as type Date is appended to the parent record (as created by the previous call to `DuxAddRecord`) and is assigned the date represented by the values in the three number objects. Note that Generate does not perform any validation on date values.
Returns	`DUXRC_InvalidName`: the field is invalid for the parent record. `DUXRC_InvalidType`: the field's definition is not compatible with a date.

DuxAddNumberField Function

Function:	Adds a field of type `Number` to a `Data Input` record and assigns a value.
Prototype:	`DUXRC DuxAddNumberField (HDUX <hdux>, PDUXPARENT <pParent>, PDUXSTRING <pstrName>, PDUXNUMBER <pnumVal>)`
Parameters:	`hdux handle` to the user exit service. `pParent` pointer to parent record. `pstrName` string; name of the counter field to be added. `pnumVal` pointer to number object.
Description	This function is used only for Data Input user exits. A field defined as type Number (signed or unsigned decimal number with floating point) is appended to the parent record (as created by the previous call to `DuxAddRecord`) and is assigned the value in the number object.
Returns	`DUXRC_InvalidName`: the field is invalid for the parent record. `DUXRC_InvalidType`: the field's definition is not compatible with a number.

DuxAddRecord Function

Function:	Adds a record to the current Data Input publication being prepared
Prototype:	`PDUXPARENT DuxAddRecord (HDUX <hdux>, PDUXPARENT <pParent>, PDUXSTRING <pstrKey>)`
Parameters:	`hdux` handle to the user exit service. `pParent` pointer to parent record. `pstrName` string; key for the record to be added.
Description	This function is used only for Data Input user exits. A record with the key name indicated is added to the publication data being prepared in response to a `DUXDA_ReadPublication` request. Unless this is the first record to be added (and is therefore not the 'start of publication' record) you should indicate this records parent.
Returns	This function returns the pointer to the record to be added. If NULL is returned, either the definition matching the specified record key was not found, or the record is not a valid child of the parent record.

DuxAddStringField Function

Function:	Adds a field of type String to a `Data Input` record and assigns a value.
Prototype:	`DUXRC DuxAddStringField (HDUX <hdux>, PDUXPARENT <pParent>, PDUXSTRING <pstrName>, PDUXSTRING <pstrVal>)`
Parameters:	`hdux` handle to the user exit service. `pParent` pointer to parent record. `pstrName` string; name of field to be added.. `pstrVal` string; pointer to string field to be added.
Description	This function is used only for Data Input user exits. A field defined as type String is appended to the parent record (as created by the previous call to DuxAddRecord) and is assigned the value in the string object.
Returns	`DUXRC_InvalidName`: field name is invalid for parent record.

DuxGetCodePage Function

Function:	Gets the name of the code page currently set.
Prototype:	`DUXCP DuxGetCodePage (HDUX <hdux>)`
Parameters:	`hdux` handle to the user exit service.
Description	Generate interprets strings using a code page. The range of supported code pages are defined as constants in the user exit header file. The default code page is assumed from the production platform but you can change this using `DuxSetCodePage`. This function returns the name of the code page (the header file constant) that is currently being used.
Returns	This function returns the code page identifier.

DuxGetFileByteBuffer Function

Function:	Gets pointer to Generate file buffer.
Prototype:	`PDUXBYTE DuxGetFileByteBuffer (HDUX <hdux>)`
Parameters:	`hdux` handle to the user exit service.
Description	Used in conjunction with `DuxGetFileByteDataSize` to access binary data in the Generate buffer within a File user exit. Note that the value of the pointer should not be saved and re-used as it may change between invocations of the user exit or whenever `DuxSetFileByteDataSize` is called. Care should be taken to avoid writing beyond the number of bytes returned by `DuxGetFileByteDataSize`.
Returns	This function returns a pointer to the Generate file buffer. 0 (zero) is returned when binary mode was not specified in the `DuxRegisterFile…Exit` function, when the exit is not of type File, or where the function is called in response to a request other than `DUXFA_Read` or `DUXFA_Write`.

DuxGetFileByteDataSize Function

Function:	Gets size of file buffer used by Generate.
Prototype:	`DUXINT32 DuxGetFileByteDataSize (HDUX <hdux>)`
Parameters:	`hdux` handle to the user exit service.
Description	Used in conjunction with `DuxGetFileByteBuffer` to binary data in the Generate buffer within a File user exit.
Returns	This function returns the size of the Generate file buffer. 0 (zero) is returned when binary mode was not specified in the `DuxRegisterFile…Exit` function, when the exit is not of type File, or where the function is called in response to a request other than `DUXFA_Read` or `DUXFA_Write`.

DuxGetFileOffset Function

Function:	Returns current file offset to Generate.
Prototype:	`DUXRC DuxGetFileOffset (HDUX <hdux>, PDUXUINT32 <pofs>)`
Parameters:	`hdux` handle to the user exit service. `pofs` pointer to offset value.
Description	Used with File user exits to respond to a `DUXFA_Tell` request from Generate.
Returns	`DUXRC_OK DUXRC_OutOfContext`: the function was called from a user exit not of type File or in response to a request other than `DUXFA_Tell`.

DuxGetFileOpenMode Function

Function:	Gets file open flags set by caller of user exit.
Prototype:	`DUXRC DuxGetFileOpenMode (HDUX <hdux>, PDUXUINT32 <pnFlags>)`
Parameters:	`hdux` handle to the user exit service. `pnFlags` pointer to variable which contains contains file access attributes.
Description	Used with File user exits. This function gives access to the file settings specified in the `flFlags` parameter of the relevant `DuxRegisterFile…Exit` function.
Returns	`DUXRC_OK DUXRC_OutOfContext`: the function was called from a user exit not of type File or in response to a request other than `DUXFA_Tell`.

DuxGetInvokeParameter Function

Function:	Gets a parameter from the `USEREXIT` job command.
Prototype:	`DUXRC DuxGetInvokeParameter (HDUX hdux, PDUXSTRING pstr, UINT nParam)`
Parameters:	`hdux` handle to the user exit service. `pstr` pointer to a string object in which the parameter will be stored. `nParameter` sequence number of the required parameter.
Description	You can pass parameters to the user program by extending the parameters in the appropriate `USEREXIT` command within the OPS file with which Generate was launched. Use this function to access these parameters. The reference name of the user exit is excluded from the available parameters so a sequence number of 1 returns the second parameter and so on. Use in conjunction with `DuxGetInvokeParameterCount` if you are unsure of the number of available parameters.
Returns	`DUXRC_OK` `DUXRC_BadParameter: nParameter` is out of range.

DuxGetInvokeParameterCount Function

Function:	Gets the number of parameters specified in the `USEREXIT` job command.
Prototype:	`DUXUINT32 DuxGetInvokeParameterCount (HDUX hdux)`
Parameters:	`hdux` handle to the user exit service.
Description	You can pass parameters to the user program by extending the parameters in the appropriate USEREXIT command within the OPS file with which Generate was launched. Use this function to identify the number of available parameters. Use in conjunction with DuxGetInvokeParameter.
Returns	Number of parameters specified.

DuxGetOpsSymbolValue Function

Function:	Gets symbol value is stored from the Generate OPS file.
Prototype:	`BOOL DuxGetOpsSymbolValue (HDUX hdux, PDUXSTRING pstrName, PDUXSTRING`
Parameters:	`hdux` handle to the user exit service. `pstrName` pointer to DUXSTRING containing name. `pstrValue` pointer to DUXSTRING where value is stored. `pstrDefault` pointer to DUXSTRING to use as default value if symbol is undefined. If NULL, empty string will be returned as default.
Description	You can pass parameters to the user program by extending the parameters in the appropriate `USEREXIT` command within the OPS file with which Generate was launched. Use this function to identify the number of available parameters. Use in conjunction with `DuxGetInvokeParameter`.
Returns	Number of parameters specified.

DuxInformUser Function

Function:	Outputs an information message through Generate.
Prototype:	`Void DuxInformUser (HDUX hdux, PDUXSTRING pstrMsg)`
Parameters:	`hdux` handle to the user exit service. `pstrMsg` pointer to message to display.
Description	The exact mechanism used by Generate to output the message depends on platform and configuration. The message is defined as type 'Information' to Generate and therefore does not trigger any exception handling.
Returns	`DUXRC_OK`

DuxNumAlloc Function

Function:	Allocates a number object.
Prototype:	`PDUXNUMBER DuxNumAlloc (HDUX hdux)`
Parameters:	`hdux` handle to the user exit service.
Description	You must create a number object to hold numeric data that needs to be passed between Generate and the user exit program.
Returns	`DUXRC_OK`

DuxNumFree Function

Function:	Releases a number object.
Prototype:	`PDUXNUMBER DuxNumFree (HDUX hdux, PDUXNUMBER pnum)`
Parameters:	`hdux` handle to the user exit service. `pnum` pointer to number object.
Description	The memory associated with the object is freed.
Returns	`DUXRC_OK`

DuxNumFromINT Function

Function:	Converts an integer for use in a User Exit.
Prototype:	`VOID DuxNumFromINT (HDUX hdux, PDUXNUMBER pnum, DUXINT32 nint);`
Parameters:	`hdux` handle to the user exit service. `pnum` pointer to number object. `nint` the integer to be converted
Description	Converts an integer to the internal number format (`DUXNUMBER`) used by the User Exit Interface.
Returns	Nothing

DuxNumStore Function

Function:	Stores a value in a number object.
Prototype:	`DUXRC DuxNumStore (HDUX hdux, PDUXNUMBER pnum, PDUXSTRING pstrWhole, PDUXSTRING pszFraction)`
Parameters:	`hdux` handle to the user exit service. `pnum` pointer to number object. `pstrWhole` whole part of number. Spaces, commas and decimal points are ignored. Maximum length is 24 digits. `pszFraction` fraction part of number. Maximum length is 8 digits.
Description	The value must be coded using two separate strings, one to represent the whole and one to represent the fractional part of the number. Both strings must convert to an actual numbers value.

Returns	`DUXRC_OK`
	`DUXRC_InvalidWhole`
	`DUXRC_InvalidFraction`

DuxRaiseWarning Function

Function:	Outputs a warning message through Generate.
Prototype:	`Void DuxRaiseWarning (HDUX hdux, PDUXSTRING pstrMsg)`
Parameters:	`hdux` handle to the user exit service. `pstrMsg` pointer to message to display.
Description	The exact mechanism used by Generate to output the message depends on platform and configuration. The message is defined as type 'Warning' to Generate and therefore does not trigger any exception handling.
Returns	`DUXRC_OK`

DuxRegisterDataInputExit Function

Function:	Registers a data input user exit.
Prototype:	`DUXRC DuxRegisterDataInputExit (HDUX hdux, PDUXSTRING pstrName, PDUXDATAINPUTFN pfn, void* pvUser)`
Parameters:	`hdux` handle to the user exit service. `pstrName` name of the user exit. It may not contain the characters ' (' or')' . `pfn` pointer to function that Generate will use to call exit. `pvUser` pointer to user data. The pointer is passed back to the user when 'pfn' is called.
Description	When coding a Data Input user exit you must call this function within the `DOC1REG` portion of the user program.

Returns	`DUXRC_OK`
	`DUXRC_InvalidWhole`
	`DUXRC_InvalidParameter`

DuxRegisterFileInputExit Function

Function:

Registers a file input user exit.

Prototype:

```
DUXRC DuxRegisterFileInputExit (HDUX hdux,
PDUXSTRING pstrName, DUXUINT32 flFlags,
PDUXFILEFN pfn, void* pvUser)
```

Parameters:

hdux handle to the user exit service.

pstrName name of the user exit. It may not contain the characters ' (' or')' .

flFlags this must be passed as a concatenation of constants (as defined in the API header file) that together indicate the file contents and access method.

Mode	you must include the constant DUFF_Read in this function.
Type	include DUFF_Record to indicate record input or DUFF_Stream for stream input.
Format	include DUFF_Binary to indicate binary contents or DUFF_Text for text.

pvUser pointer to user data. The pointer is passed back to the user when pfn is called.

Description

When coding a File user exit to replace an input file you must call this function within the DOC1REG portion of the user program.

Returns

DUXRC_OK

DUXRC_InvalidWhole

DUXRC_InvalidParameter

DuxRegisterFileOutputExit Function

Function:	Registers a file output user exit.
Prototype:	`RegisterFileOutputExit (hdux, pstrName, flFlags, pfn, pvUser)`
Parameters:	`hdux` handle to the user exit service. `pstrName` name of the user exit. It may not contain the characters ' (' or')' . `flFlags` this must be passed as a concatenation of constants (as defined in the API header file) that together indicate the file contents and access method. **`Mode`** you must include the constant `DUFF_Read` in this function. **`Type`** include `DUFF_Record` to indicate record input or `DUFF_Stream` for stream input. **`Format`** include `DUFF_Binary` to indicate binary contents or `DUFF_Text` for text. `pvUser` pointer to user data. The pointer is passed back to the user when `pfn` is called.
Description	When coding a File user exit to replace an output file you must call this function within the `DOC1REG` portion of the user program.

Returns	`DUXRC_OK`
	`DUXRC_InvalidWhole`
	`DUXRC_InvalidParameter`

DuxRegisterJobEndingEvent Function

Function:	Identifies a function to be called when the Generate job ends.
Prototype:	`DUXRC DuxRegisterJobEndingEvent (HDUX hdux, PDUXJOBENDINGFN pfn, void*`
Parameters:	`hdux` handle to the user exit service. `pfn` pointer to function to be called. `pvUser` pointer to user data. The pointer is passed back to the user when `pfn` is called.
Description	Code this function in the `DOC1REG` portion of the user program to identify a function to be called immediately before a Generate job ends – i.e. when all input data has been processed. In memory resident versions of Generate the function will be called every time a batch of data is processed. To identify a function that is only used when the Generate process is fully closed use `DuxRegisterTerminateEvent`.
Returns	`DUXRC_OK`

DuxRegisterJobStartingEvent Function

Function:	Identifies a function to be called when the Generate job starts.
Prototype:	`DUXRC DuxRegisterJobStartingEvent (HDUX hdux, PDUXJOBSTARTINGFN pfn, void*`
Parameters:	`hdux` handle to the user exit service. `pfn` pointer to function to be called. `pvUser` pointer to user data. The pointer is passed back to the user when `pfn` is called.
Description	Details Code this function in the `DOC1REG` portion of the user program to identify a function to be called immediately before a Generate job starts. In memory resident versions of Generate note that this function is only called once when the process is initiated.
Returns	`DUXRC_OK`

12 - Structured XML journals

Journals are used to record the activity of publications when processed in the production environment. This type of journal is XML based and complies to a predefined structure. Refer to the Designer User's Guide for further information on using Structured XML journals.

Note that structured XML journals have a reserved Publish/Generate OPS alias of:

```
<Journal>SYS_XPJ=filename
```

In this section

<ProductionJournal>

Description:	Journal root element
Attributes	version version number. runtype Identifies the Generate run type. Valid values are: EOBatch, EONeutral, Batch.
Contents	CompositionDate - one only. OutputDevices - one only. StartOfJob - optional, single occurrence if included Publications - one only EndOfJob - optional, single occurrence if included
Example	<ProductionJournal>…</ProductionJournal>

<CompositionDate>

Description:	Mandatory date of composition run.
Attributes	None.
Contents	Date
Example	`<CompositionDate>2009-11-02</CompositionDate>`

<OutputDevices>

Description:	Mandatory output devices element.
Attributes	None.
Contents	One or more output device elements for each output datastream.
Example	`<OutputDevices>…</OutputDevices>`

<StartOfJob>

Description:	Optional start of job journal entry. Note that this journal entry is set at publication level in the Designer
Attributes	None
Contents	`JE` - one or more.
Example	`<StartOfJob>…</StartOfJob>`

<JE>

Description:	Journal entry
Attributes	`name` - journal entry name `type` -data type, as configured in the publication logic
Contents	None
Example	`<JE type="dat" name="StartOfJobJournal">2009-11-02</JE>`

<Publications>

Description:	Publication root element
Attributes	None
Contents	`Pub` - one or more.
Example	`<Publications>…</Publications>`

<Pub>

Description:	Publication element
Attributes	`name` - publication logic map label `instanceId` - document instance ID
Contents	`PBO` - one or more `PBC` - one or more
Example	`<Pub name="Pub1" instanceId="AD0F4956BAE7044391CEB6FA4D30B179">`

<PBO>

Description:	Publication offset
Attributes	`idx` - output device index number `totalPages` - total physical page count `totalPagesRecto` - front-side page count `filename` - output file containing publication
Contents	None
Example	`<PBO idx="1" totalPages="8" totalPagesRecto="4" filename="out.pdf">0</PBO>`

<PBC>

Description:	Publication content component
Attributes	None
Contents	`JE` - one or more `Doc` - one or more
Example	`<PBC>…</PBC>`

<Doc>

Description:	Each document in the publication is represented by a Doc element
Attributes	`name` - the name applied to the document in the publication logic
Contents	`DO` - one or more `PG` - one or more
Example	`<Doc>…</Doc>`

<DO>

Description:	Document offset element. Contains the offset of the document from the start of the output datastream
Attributes	`idx` - output device index number
Contents	None
Example	`<DO idx="2">324922</DO>`

<PG>

Description:	Page entry. A single page entry element exists for each page in the document.
Attributes	None
Contents	`PGO` - one or more `PGC` - optional, single occurrence
Example	`<PG>…</PG>`

<PGO>

Description:	Page offset. This element the offset of the page from the start of the output datastream. A single PGO element exists for each output device.
Attributes	`idx` - output device index number `jpn` - job page number `ppn` - publication page number `dpn` - document page number
Contents	None
Example	`<PGO idx="1" jpn="1" ppn="1" dpn="1">72</PGO>`

<PGC>

Description:	Publication content component
Attributes	None
Contents	`idx` - one or more `Doc` - one or more
Example	`<PGC>…</PGC>`

<JE>

Description:	Journal entry
Attributes	`name` - journal entry name `type` - data type, as configured in the publication logic
Contents	None
Example	`<JE type="int" name="AC2">7</JE>`

<EndOfJob>

Description:	End of job journal entries
Attributes	None
Contents	`JE` - one or more
Example	`<EndOfJob>…</EndOfJob>`

<JE>

Description:	Optional start of job journal entry. Note that this journal entry is set at publication level in the Designer
Attributes	`name` - identifies the output datastream `type` - data type, as configured in the publication logic
Contents	None
Example	`<JE type="dat" name="StartOfJobJournal">2009-11-02</JE>`

Example

```
extern "C" __declspec(dllexport) DUXRC DOC1REG (HDUX hdux)
{
PDUXSTRING pstr = DuxStrAlloc (hdux);
DUXINT32 nFlags;
//
// Set the flags indicating the type of IO we support. This user exit just
// supports reading of record mode interface.
//
//
nFlags = DUXFF_Read | DUXFF_RecordMode | DUXFF_Binary;
//
// Register our File Input User Exit with the name 'BWFI'. It doesn't need to
// be the same name as the DLL. It just happens to be.
//
DuxStrSetString (hdux, pstr, "BWFI");
DuxRegisterFileInputExit (hdux, pstr, nFLags, MyFileInputExit, mydata);
//
// Register when Generate terminates so I know when to free 'mydata'.
//
DuxRegisterTerminateEvent (hdux, FreeMyData, mydata);
//
// Let the user know that we've been loaded.
//
DuxStrSetString (hdux, pstr, "Boardwalk File Input Interface ready for use.");
DuxInformUser (hdux, pstr);
DuxStrFree (hdux, pstr);
return DUXRC_OK;
}
extern "C" DUXRC MyFileInputExit (HDUX hdux, DUXFILEACTION action, void *pvUser)
{
DUXRC rc = DUXRC_OK;
switch (action)
//
// Generate is requesting the file to be opened.
//
case DUXFA_Open:
{
PMYDATA pmydata = (PMYDATA)pvUser.
//
// Ensure there are two parameters specified when we are invoked.
// If not, tell the user what they need to do.
//
if (DuxGetInvokeParameterCount (hdux) < 2)
{
...
rc = DUXRC_Failed;
}
//
// Otherwise we’ll open up the two files specified and raise an error
// if we can’t find them.
//
else
{
PDUXSTRING pstrFile1 = DxGetInvokeParameter (hdux, 1);
PDUXSTRING pstrFile2 = DxGetInvokeParameter (hdux, 2);
...
}
}
break;
//
// Generate is requesting to read a record from the file.
//
case DUXDA_Read:
{
FILE *pfToUse = null;
// Figure out which file to use for the read.
if (!feof (pmydata->pfFile1))
pfToUse = pmydata->pfFile1;
else if (!feof (pmydata->pfFile2))
pfToUse = pmydata->pfFile2;
// If we have a file with data in it...
if (pfToUse != NULL)
{
char *pchEol = NULL;
// Read data from file.
fgets (pmydata->szBuffer, sizeof (pmydata->szBuffer)-1, pfToUse);
// Remove the end of line character. Generate doesn’t want it and will
// treat it as data.
puchEol = strchr (pmydata->szBuffer, '\n');
if (puchEol != NULL)
 *puchEol = '\0';
// Give Generate the record data.
DuxSetFileData (hdux, pmydata->szBuffer, strlen (pmydata->szBuffer));
}
// Otherwise we don’t have anymore data.
else
rc = DUXRC_EndOfData;
break;
//
// Generate is asking if there is any more data to read.
//
case DUXDA_IsEndOfData:
if (feof (pmydata->pfFile1) && feof (pmydata->pfFile2))
rc = DUXRC_EndOfData;
else
rc = DUXRC_DataAvailable;
break;
//
// Generate is requesting that the file be closed.
//
case DUXDA_VirtualClose:
{
PDUXSTRING pstr = DuxStrAlloc (hdux);
fclose (pmydata->pfFile1);
fclose (pmydata->pfFile2);
DuxStrSetString (hdux, pstr, "Boardwalk File Input Exit Closed");
DuxInformUser (hdux, pstr);
DuxStrFree (hdux, pstr);
}
break;
*/
}
return rc;
}
```

13 - Output datastream formats

By default, Generate produces an output datastream with a file and record structure considered to be the most suitable for its type and the platform on which it was produced. You may need to adjust these structures to suit your output environment particularly if datastreams need to be transferred to other platforms or pass through additional media before actually being printed/presented on the output device.

In this section

Working with Designer output formats

Designer has a range of predefined output datastream formats that cater for the most common of these requirements. These are specified by selecting or entering the relevant keyword plus any required parameters either within an Output Device object in the Designer or as part of an open command in a PCE script. If the predefined formats do not produce the file and record structures suitable for your environment you can use specific formatting codes to indicate the precise structure required. These are coded in place of the regular keywords as above. See “Customizing output formats”.

Note: For more information about where to use the keywords or formatting codes: see “open” on page 135 for PCE jobs; for Generate see the information about output device objects in the Designer User’s Guide.

Predefined output formats

standard

No special file blocking or record structures. This equates to formatting codes:

```
$BN($RV($CC,$PD))
```

barrpc

Suitable for Xerox 'online' emulation via a BARR server with output produced by Generate on a PC host. This equates to formatting codes:

```
$BN($RV($PS(M,2,9),$HV(00,00),$PS(M,2,5),$HV(00,00),$CC,$PD))
```

crlf or line

Each record is ended with the standard ASCII text record ‘new line’ format (carriage return/line feed). This equates to formatting codes:

```
$BN($RV($CC,$PD,$HV(0D,0A)))
```

ksdsafp (length)

Used for AFPDS records within a VSAM KSDS table under z/OS. length indicates the record length of the recipient dataset. This equates to formatting codes:

```
$BK(80,00,$PD)
```

KSDS notes

This format is supported for AFP output only. Suitable KSDS datasets must be defined as type 'INDEXED' with a key length of 10 bytes starting in the first byte of the record. The following is an example of the Define Cluster command syntax that would generate a KSDS dataset supported by Designer (assumes a length of 1000):

```
DEFINE CLUSTER -
NAME(MYUSER.DOC.KSDS) -
INDEXED -
KEYS(10 0) -
SPEED -
SHR(1 3) -
FREESPACE(0 10) -
RECORDSIZE(1010 1010)) -
DATA -
(NAME(MYUSER.DOC.KSDS.DATA) -
VOLUMES(VOLSER01) -
```

```
TRACKS(10 5) -
CONTROLINTERVALSIZE(1024)) -
INDEX -
(NAME(MYUSER.DOC.KSDS.INDEX) -
VOLUMES(USER91) -
CONTROLINTERVALSIZE(1024))
```

pp4235

Suitable for Xerox 4235 printers when attached via a parallel port. This equates to formatting codes:

$BN($RV($HV(35),$PS(L,1,1),$PD,$CC))

rdw

Each record has a 2 byte header field which stores the record length. This equates to formatting codes:

```
$BN($RV($PS(L,2,1),$CC,$PD))
```

record [(length, {T |F})]

An unblocked file with variable length records under z/OS. length optionally specifies the record length associated with the file.

- T – indicates that the record will be trimmed (the default setting)
- F – the record will not be trimmed.

If you are writing to a variable blocked dataset under z/OS you should specify a length greater than the longest possible record length to be produced (e.g. 8205 for AFPDS). For fixed blocked datasets specify the block size or greater.

rrdsafp (length)

Used for AFPDS records within a VSAM RRDS table under z/OS. length indicates the record length of the dataset. This equates to formatting codes:

```
$BR(80,00,$PD)
```

RRDS notes

This format is supported for AFP output only. Each page in the datastream always starts at the beginning of a record slot but may span multiple slots. The final slot occupied by a page is padded to the start of the next slot. Suitable RRDS datasets are defined as type 'numbered'. The following is an example of the Define Cluster command syntax that would generate an RRDS dataset supported by Designer (assumes a length of 100):

```
DEFINE CLUSTER -
(NAME(DATA.FOR.PCE) -
RECORDSIZE(100 100) -
VOLUMES(VSER01) -
TRACKS(10 5) -
NUMBERED)
```

spur

Suitable for some configurations of Xerox online emulation via a SPUR server. This equates to formatting codes:

```
$BN($RV($HV(1F,4A),$CC,$PS(M,2,0),$PD)
```

wsafp

Suitable for most AFP datastreams when the production platform is not z/OS. The AFPDS is produced as a true stream and has no special blocking or record formatting. This equates to formatting codes:

```
$BN($RV($RD))
```

Customizing output formats

The file formatting codes described in this section allow you to customize the structure of output datastream files used in the Designer environment. Typically the codes are used to define block and record header structures that supplement the output datastream data.

File Blocking

File blocking (i.e. where keywords $BV, $BR or $BF are used) relates to the use of Block Descriptor Words (BDWs) to separate the output datastream into logical packets of user defined size. Most typically blocking is used to generate an interim file format compatible with data transfer software such as that used with some tape devices. If your output does not require blocking specify the $BN keyword.

> **Note:** On z/OS system the logical file blocking provided by Generate supplements any blocking provided by the native features of the operating system. If file blocking is specified the block controls will be created in addition to the those generated by the operating system.

Logical Records

An output datastream is considered to be made up of logical records. The structure of such records and the amount of data they contain depends on the type of output datastream being used:

For:	a logical record is:
AFP	the data that makes up a single AFPDS structured field.

PCL	the escape sequence(s) required for a single print operation (write line, write text, etc.).

As with file blocking, Generate's concept of logical records should not be confused with the physical record constructs used by z/OS. However, you should be aware of the default methods of how the generated output datastreams work within such physical records.

> **Note:** Specifically under z/OS For AFPDS and LineData, all logical records are written as individual physical records; for other datastreams all logical records are written as a continuous stream, i.e. they can span individual physical records. For all datastream types other than LineData, all logical records are written as a continuous stream; for LineData all logical records are written as individual physical records.

Command Syntax

Formatting codes are a sequence of keywords and associated parameters that fall into four categories:

- ***Output Description*** and ***Record keywords*** - contain control information for (respectively) the blocks and records making up the file.
- ***Block Data*** and ***Record Data*** keywords - define the actual structure of a output datastream file, i.e. the relative positions of carriage controls, block and record headers and the actual output datastream logical records themselves (the $PD keyword).

The syntax of formatting codes is illustrated in the diagram on the following page. Each keyword can be followed by parameters enclosed in parenthesis. The order of parameters must match that specified in the diagram. Keywords and parameters must always be separated by commas.

File blocking

File Blocking

File blocking (i.e. where keywords $BV, $BR or $BF are used) relates to the use of Block Descriptor Words (BDWs) to separate the output datastream into logical packets of user defined size. Most typically blocking is used to generate an interim file format compatible with data transfer software such as that used with some tape devices. If your output does not require blocking specify the $BN keyword.

> **Note:** On z/OS system the logical file blocking provided by Generate supplements any blocking provided by the native features of the operating system. If file blocking is specified the block controls will be created in addition to the those generated by the operating system.

Logical records

An output datastream is considered to be made up of logical records. The structure of such records and the amount of data they contain depends on the type of output datastream being used:

For:	a logical record is:
AFP	the data that makes up a single AFPDS structured field.
PCL	the escape sequence(s) required for a single print operation (write line, write text, etc.).

As with file blocking, Generate's concept of logical records should not be confused with the physical record constructs used by z/OS. However, you should be aware of the default methods of how the generated output datastreams work within such physical records.

> **Note:** Specifically under z/OS For AFPDS and LineData, all logical records are written as individual physical records; for other datastreams all logical records are written as a continuous stream, i.e. they can span individual physical records. For all datastream types other than LineData, all logical records are written as a continuous stream; for LineData all logical records are written as individual physical records.

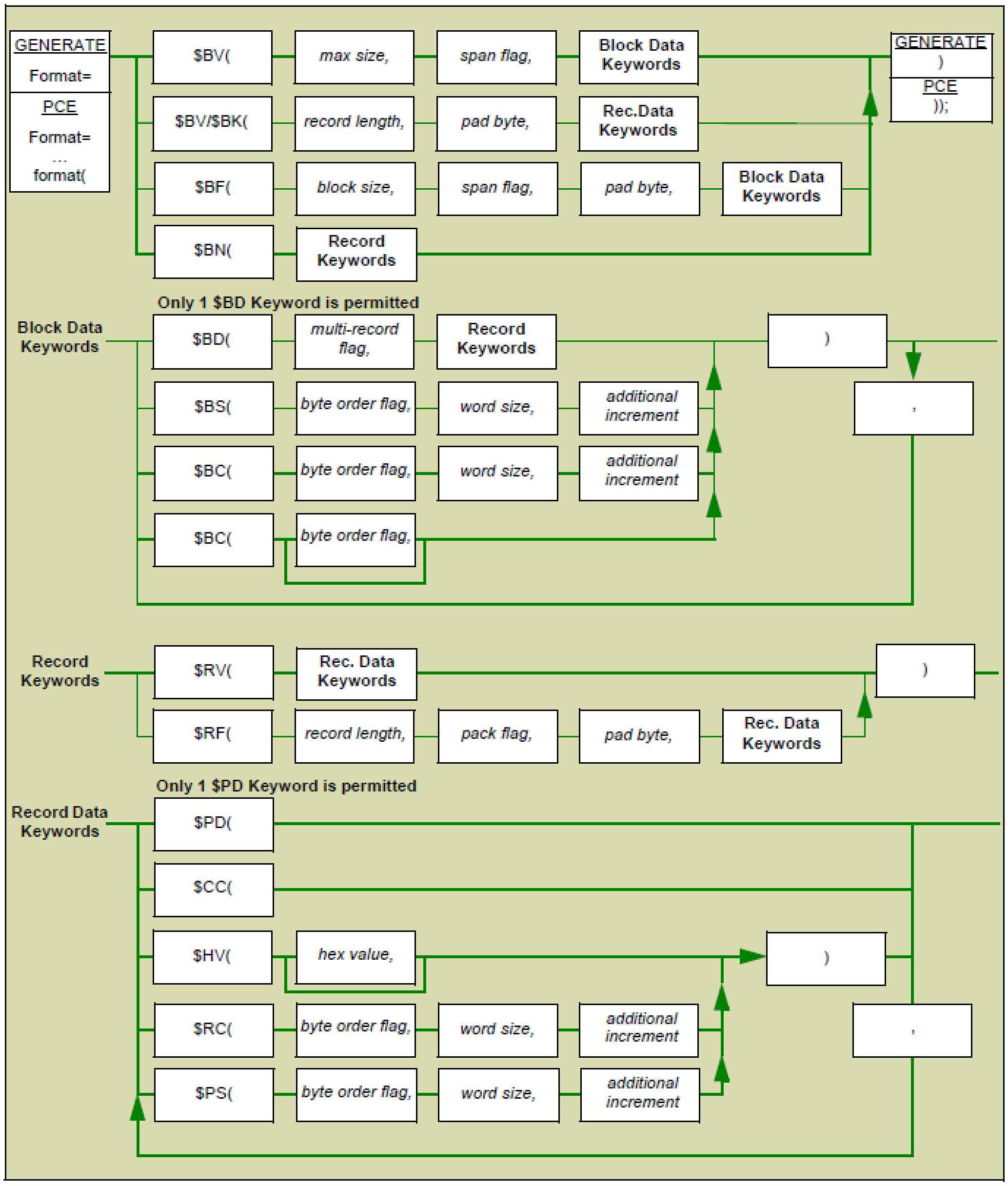
Command Syntax

Formatting codes are a sequence of keywords and associated parameters that fall into four categories:

- ***Output Description*** and ***Record keywords*** - contain control information for (respectively) the blocks and records making up the file.
- ***Block Data*** and ***Record Data*** keywords - define the actual structure of a output datastream file, i.e. the relative positions of carriage controls, block and record headers and the actual output datastream logical records themselves (the $PD keyword).

The syntax of formatting codes is illustrated in the diagram on the following page. Each keyword can be followed by parameters enclosed in parenthesis. The order of parameters must match that specified in the diagram. Keywords and parameters must always be separated by commas.

Command syntax schematic



Keywords

Output Description

Output description keywords are mutually exclusive and you must code one of the following:

$BN – unblocked data

The file does not use BDWs to separate data packets. This keyword must be followed by Record keywords as appropriate.

$BV – variable blocked data The file includes BDWs at the beginning of each data packet. The data packets are of variable size. This keyword must be followed by max size and span flag parameters in sequence followed by Block Data keywords as appropriate.

$BF – fixed block data The file includes BDWs at the beginning of each data packet. The data packets commence at fixed offsets from the start of file. This keyword must be followed by block size, span flag and pad byte parameters in sequence followed by Block Data keywords as appropriate.

$BR – VSAM RRDS data The file is stored as a VSAM relative record dataset (RRDS) under z/OS. The dataset has fixed length records of size record length. See “RRDS notes” on page 322 for more information. This keyword must be followed by record size and pad byte parameters in sequence followed by Record Data keywords as appropriate.

$BK – VSAM KSDS data The file is stored as a VSAM keyed sequence dataset (KSDS) under z/OS. The dataset has fixed length records of size record length. See “KSDS notes” on page 321 for more information. This keyword must be followed by record size and pad byte parameters in sequence followed by Record Data keywords as appropriate.

Block Data

For file types $BN and $BV you must code one $BD keyword. You may also code one or more $BS, $BC and $HV keywords as appropriate to the file. The order of these keywords to match a typical block data construct will be any occurrences of $BS, $BC and $HV (typically making up the file header information) followed by $BD. This is not a restriction however and you may code the keywords in any order.

$BS – block size. Each block has a field that contains its size (in bytes). This keyword must be followed by byte order flag, word size and additional increment parameters in sequence.

$BC – block sequence. Each block has a field containing a sequence number. When generated by Generate this sequence number will start at 1 and be incremented by 1 for each block. This keyword must be followed by byte order flag, word size and additional increment parameters in sequence.

$HV – additional header data. Each block has the hex value(s) specified as additional header data. Each hex value specified represents a byte in the header information. Multiple values are separated by commas.

$BD – block data. This keyword marks the relative position of the data contained in the block itself (as distinct from the fields/data indicated by other keywords in the block data section). The $BD keyword must be followed by the *multiple record flag* parameter followed by a Record keyword as appropriate.

Record

All file descriptions must contain either the $RF or $RV keywords.

$RV – variable length records. The file has records of individual length appropriate for the amount of data stored. No parameters are required or allowed. This keyword must be followed by Record Data keywords as appropriate.

$RF – fixed length records. All records in the file are the same length. The record length is explicitly specified. This keyword must be followed by record size, pack flag and pad byte parameters in sequence followed by Record Data keywords as appropriate.

Record Data

You must code the $PD keyword for all file types. You may also code one or more $CC, $PS, $RC and $HV keywords as appropriate to the file.

$PD – output datastream record. This keyword marks the relative position of the output datastream logical record itself (as distinct from any carriage controls or other fields specified as record data).

$CC – carriage control. This keyword marks the relative position of a carriage control byte and should only be coded for output datastream using such controls, i.e. LineData. If this is coded for an output datastream to be generated by Generate but the output datastream does not support carriage controls it is ignored.

$PS – record size. Each record has a field that contains the size (in bytes) of the output datastream record. Note that this value will not include the size of any additional data that has been added via $CC and $HV keywords unless you specifically include it as the additional increment value. This keyword must be followed by byte order flag, word size and additional increment parameters in sequence. Also see span flag parameter below.

$RC – record sequence. Each record has a field containing a sequence number. When generated by Generate this sequence number will start at 1 and be incremented by 1 for each record. This keyword must be followed by byte order flag, word size and additional increment parameters in sequence.

$HV – additional data Each record has hex value(s) as an additional field at the position indicated by the sequence. Each hex value specified represents a byte in the field. Multiple values are separated by commas.

Parameters

additional increment: for output files this parameter indicates a number that will be added or subtracted from the value that would otherwise be generated as the appropriate record ($PS) or block size ($BS). The value is ignored for input files but the parameter must still be specified.

block size: the size of all blocks in a fixed block file ($BF). A value of up to 2,147,483,647 may be specified. Note that block size should not include the length of any additional increment parameters if these are also specified.

byte order flag: indicates the order in which the bytes making up the value in an additional field value are stored. If the flag is set to 'L' the word(s) are written/read in Least Significant Byte ('little endian') fashion as required by operating systems such as Windows. If the flag is set to 'M' the word(s) are written/read in Most Significant Byte ('big endian') fashion as required by operating systems such as z/OS

hex value: is a representation of one or more hex values (e.g. '0A') that will be used to supplement header data. Where multiple values are specified each must be separated by a comma. Each value specified adds one byte to the file for each logical record.

max size: the largest block size used/to be used for a variable blocked file ($BV). A value of up to 2,147,483,647 may be specified.

multiple record flag: for blocked files this flag indicates whether or not multiple records are present/permitted within a single block. If the flag is set to 'N' a single record only will be written/read from each block. If set to 'Y' records will span blocks where necessary.

pack flag: when dealing with output datastream to a fixed length record structure, this flag indicates whether or not each physical record can contain multiple datastream 'records'. If 'N' is specified a single datastream record is written/read to each physical record. If 'Y' is specified as many datastream records as possible will be written/read from each physical record.

pad byte: is a representation of a hex value (e.g. '0A') that has/will be used to pad out any bytes not used for data.

record length: as part of the $RF keyword, the number of bytes in each record. As part of the $BR or $BK keywords, the record size of a VSAM RRDS or KSDS dataset. The value specified is the equivalent of the parameters supplied to the RECORDSIZE keyword as part of the VSAM Define Cluster Command. Refer to the IBM document "VSAM Catalog Administration: Access Method Services Reference" for details of acceptable maximum values for this format.

span flag: indicates whether individual records within blocked files can span multiple blocks.

Values:

- Y – spanning is used i.e. block boundaries will be ignored when writing/reading records.

- N – spanning is not used i.e. write operations will fit as many whole records as possible within a block then restart following the next block boundary. Unused space thus created will be padded with the specified pad byte (see above).
- L – same as N except that, where the formula also includes the $PS keyword, the size field of records immediately prior to the block boundary will automatically include the number of pad bytes used. Note that this applies to $BF formulas only and not $BV formulas.

word size: the size (in bytes) of the space reserved/allocated for the appropriate parameter.

Examples

The following examples show some typical file formatting requirements and how they can be specified using Generate file formatting codes.

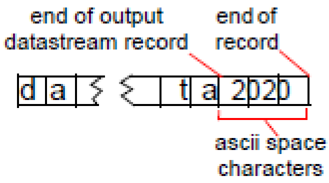
Standard

Unblocked file. Unpacked variable length records. A carriage control byte is included in position 1 if appropriate to the datastream.

```
$BN($RV($CC,$PD))
```

Fixed length

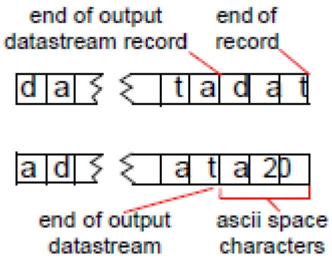
Unblocked file. Unpacked 300-byte fixed length records padded with ASCII ‘space’ characters.



```
$BN($RF(300,N,20,$PD))
```

Packed

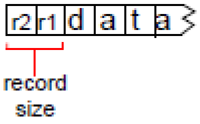
Unblocked file. Logical records are packed across 300-byte fixed length physical records and are padded with ASCII space characters.



```
$BN($RF(300,Y,20,$PD))
```

Intel or Xerox style Record Descriptor Word

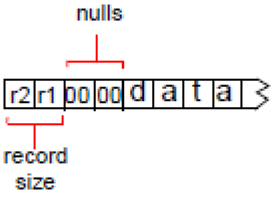
Unblocked file. Unpacked variable length records each with record size stored as a leading header field comprising 2 bytes with 'Least Significant Byte' format. No increment to the standard record size is required. A carriage control byte is included in position 1 if appropriate to the datastream. It is typically used when software controlling external media requires a header in this format.



```
$BN($RV($PS(L,2,1),$CC,$PD))
```

IBM-style Record Descriptor Word

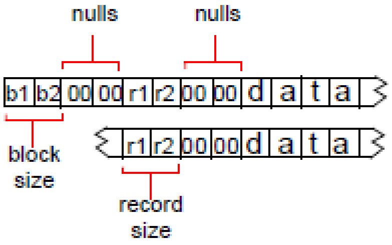
Unblocked file. Unpacked variable length records each with a header comprising a two byte record size field in 'Most Significant Byte' format plus two bytes of null padding (x’00’). The standard record size is incremented by four to account for this. No carriage control is required. This format is typically used for output datastreams generated under Windows or UNIX when software controlling external media requires a header in this format. It is not normally required in an z/OS environment.



```
$BN($RV($PS(M,2,4),$HV(00,00),$PD))
```

IBM style Block and Record Descriptor Words

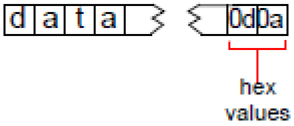
Blocked file with 8209 byte maximum block size, each with a header comprising a two byte block size field in 'Most Significant Byte' format plus two bytes of null padding (x'00'). Records are unpacked with variable length and each with a header comprising a two byte record size field in ‘Most Significant Byte’ format plus two bytes of null padding (x’00’). To account for these headers the block sizes are incremented by eight and the record sizes by four. No spanning of blocks takes place and therefore there is always one record per block. No carriage control is required.



```
$BV(8209,N,$BS(M,2,8),$HV(00,00),$BD(Y,$RV($PS(M,2,4),$HV(00,00),$PD)))
```

Carriage Return/Line Feed

Unblocked file. Unpacked variable length records each with a two-byte terminator code in the format of ASCII carriage return/line feed (x'0D', x'0A'). A carriage control byte is included in position 1 if appropriate to the datastream.



```
$BN($RV($CC,$PD,$HV(0D,0A)))
```

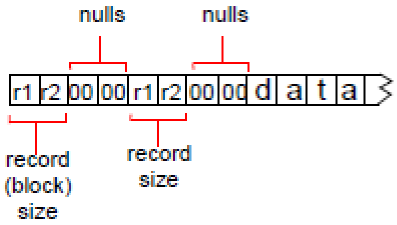
VSAM RRDS for AFP

File has VSAM blocking structure with 80-byte blocks padded with x’00’ where necessary. Blocks may store multiple records where possible. Records are variable length. No carriage control is required. Refer to your VSAM documentation for more information about such structures.

$BR(80,00,$PD)

Xerox online emulation via a BARR server

The file is unblocked in the normal sense but for the purposes of BARR software is considered to always have one record per block with the block always containing a single Generate record. Records are unpacked and of variable length. A header comprising a 2-byte record size field headers in ‘Most Significant Byte’ format plus two bytes of null padding (x’00’) appears twice with the first version acting as the ‘block’ size. The standard ‘block’ size is therefore incremented by eight bytes. Similarly, the standard record size is incremented by four bytes. A carriage control byte is included in position 1.



```
$BN($RV($PS(M,2,9),$HV(00,00),$PS(M,2,5),$HV(00,00),$CC,$PD))
```

14 - Appendix A - Generate SCP and Lookup Table codepage Overrides

This section lists the host code page numbers to be used as an override for the application data code page.

In this section

Generate SCP and lookup table override values

Generate SCP (Set Code Page) Override

Specify the required Override codepage using the number in the SCP Number column from the table below.It is important to note that code page values marked with Asterisk *, or Cardinal # indicate that these numbers have different definitions between Z/OS and Windows / UNIX or, different values between Windows/UNIX Single-Byte and Double Byte Operating Systems.

Lookup Table Override

Use the values in the Lookup table OPS values column to designate the code page association defined in the <LookupTableCodePages> of your OPS file, refer to **OPS file** on page 11 for details.

SCP Number	Lookup table OPS values	Codepage Name
0	UTF-8	Unicode (UTF-8)
12	Adobe-Koraen1-2	AdobeKor1_2 Adobe Korean1 Supplement2
14	Adobe-GB1-4	AdobeGB1_4 Adobe GB1 Supplement4
16	Adobe-Japan1-6	AdobeJap1_6 Adobe Japan1 Supplement6
37	ibm-037	Ibm037 IBM EBCDIC - U.S./Canada
273	ibm-273	Ibm273 IBM EBCDIC - Germany
277	ibm-277	Ibm277 IBM EBCDIC - Denmark/Norway
278	ibm-278	Ibm278 IBM EBCDIC - Finland/Sweden
280	ibm-280	Ibm280 IBM EBCDIC - Italy
284	ibm-284	Ibm284 IBM EBCDIC - Latin America/Spain
285	ibm-285	Ibm285 IBM EBCDIC - United Kingdom
297	ibm-297	Ibm297 IBM EBCDIC - France

SCP Number	Lookup table OPS values	Codepage Name
300	ibm-300	* Ibm300 IBM 300 - IBM DBCS Japanese
300	ibm-300-037	* Ibm300_037 IBM 300+037 - IBM MBCS Japanese
420	ibm-420	Ibm420 IBM EBCDIC - Arabic
423	ibm-423	Ibm423 IBM EBCDIC - Greek
424	ibm-424	Ibm424 IBM EBCDIC - Hebrew
437	windows-437	Win437 DOS (US)
500	ibm-500	Ibm500 IBM 500
708	windows-708	Win708 ASMO 708
720	windows-720	Win720 DOS 720 (Transparent ASMO)
737	windows-737	Win737 PC 737 (PC 437G)
775	windows-775	Win775 DOS 775
833	ibm-833	Ibm833 IBM 833 - IBM SBCS Korean
834	ibm-834	Ibm834 IBM 834 - IBM DBCS Korean
835	ibm-835	Ibm835 IBM 835 - IBM DBCS Traditional Chinese
836	ibm-836	Ibm836 IBM 836 - IBM SBCS Simplified Chinese
837	ibm-837	Ibm837 IBM 837 - IBM DBCS Simplified Chinese
838	ibm-838	Ibm838 IBM EBCDIC - Thai
850	windows-850	850 Win850 DOS 850
852	windows-852	852 Win852 PC 852

SCP Number	Lookup table OPS values	Codepage Name
855	ibm-855	Win855 IBM 855
857	ibm-857	Win857 IBM 857
858	windows-858	Win858 DOS 858 (DOS Latin 1 (Euro))
860	windows-860	Win860 DOS 860
861	windows-861	Win861 DOS 861
862	windows-862	Win862 DOS 862
863	windows-863	Win863 DOS 863
864	windows-864	Win864 Arabic (DOS 864)
865	windows-865	Win865 DOS 865
866	windows-866	Win866 DOS 866
869	windows-869	Win869 DOS 869
870	ibm-870	Ibm870 IBM EBCDIC - 870 Multilingual/ROECE (Latin-2)
871	ibm-871	Ibm871 IBM EBCDIC - Icelandic
874	windows-874	Win874 Windows 874 (DOS 8
875	ibm-875	Ibm875 IBM EBCDIC - Greek (Modern)
880	ibm-880	Ibm880 IBM EBCDIC - Cyrillic (Russian)
891	ibm-891	Ibm891 IBM 891 - IBM SBCS Korean
903	ibm-903	Ibm903 IBM 903 - IBM SBCS Simplified Chinese
904	ibm-904	Ibm904 IBM 904 - IBM SBCS Traditional Chinese

SCP Number	Lookup table OPS values	Codepage Name
905	ibm-905	Ibm905 IBM EBCDIC - Turkish (Latin-3)
926	ibm-926	Ibm926 IBM 926 - IBM DBCS Korean
927	ibm-927	Ibm927 IBM 927 - IBM DBCS Traditional Chinese
928	ibm-928	Ibm928 IBM 928 - IBM DBCS Simplified Chinese
930	ibm-930	Ibm930 IBM 930 - IBM MBCS Japanese (290+300)
932	ibm-932	# Ibm932 IBM 932 - IBM MBCS Japanese (897+301)
932	windows-932	* Win932 Windows 932 (SHIFTJIS)
933	ibm-833	Ibm933 IBM 933 - IBM MBCS Korean (834+833)
934	ibm-834	Ibm934 IBM 934 - IBM MBCS Korean (891+926)
936	ibm-936	* Ibm936 IBM 936 - IBM MBCS Simp. Chinese (903+928)
936	windows-936	* Win936 Windows 936 (GB2312)
938	ibm-938	Ibm938 IBM 938 - IBM MBCS Trad. Chinese (904+927)
939	ibm-939	Ibm939 IBM 939 - IBM MBCS Japanese (1027+300)
943	ibm-943-2000	* Ibm943_2000 IBM 943 (2000) - IBM MBCS Japanese (1041+941)
943	ibm-943-2003	* Ibm943_2003 IBM 943 (2003) - IBM MBCS Japanese (1041+941)
944	ibm-944	Ibm944 IBM 944 - IBM MBCS Korean
946	ibm-946	Ibm946 IBM 946 - IBM MBCS Simplified Chinese
949	windows-949	Win949 Windows 949 (Korean)
950	ibm-950	* Ibm950 IBM 950 - IBM MBCS Traditional Chinese

SCP Number	Lookup table OPS values	Codepage Name
950	windows-950	* Win950 Windows 950 (No UDC)
950	windows-950-udc	* Win950_UDC Windows 950 (UDC)
950	windows-950-hkscs	* Win950hkscs Windows 950 + HKSCS
951	ibm-951	Ibm951 IBM 951 - IBM DBCS Korean
954	ibm-954	Ibm954 IBM 954 - EUC-JP
955	ibm-955	Ibm955 IBM 955 - IBM JIS X 208
955	ibm-955-L1	Ibm955_L1 IBM 955 - IBM JIS X 208 (+Latin1)
970	ibm-970	Ibm970 IBM 970 - EUC-KR
1025	ibm-1025	Ibm1025 IBM EBCDIC - Cyrillic (Serbi)
1026	ibm-1026	Ibm1026 IBM EBCDIC - Turkish (Latin-5)
1043	ibm-1043	Ibm1043 IBM 1043 - IBM SBCS Traditional Chinese
1047	ibm-1047	Ibm1047 IBM EBCDIC - Latin 1/Open Systems
1088	ibm-1088	Ibm1088 IBM 1088 - IBM SBCS Korean
1140	ibm-1140	Ibm1140 IBM EBCDIC - U.S./Canada (Euro)
1141	ibm-1141	Ibm1141 IBM EBCDIC - Germany (Euro)
1142	ibm-1142	Ibm1142 IBM EBCDIC - Denmark/Norway (Euro)
1143	ibm-1143	Ibm1143 IBM EBCDIC - Finland/Sweden (Euro)
1144	ibm-1144	Ibm1144 IBM EBCDIC - Italy (Euro)
1145	ibm-1145	Ibm1145 IBM EBCDIC - Latin America/Spain (Euro)

SCP Number	Lookup table OPS values	Codepage Name
1146	ibm-1146	Ibm1146 IBM EBCDIC - United Kingdom (Euro)
1147	ibm-1147	Ibm1147 IBM EBCDIC - France (Euro)
1148	ibm-1148	Ibm1148 IBM EBCDIC - International (Euro)
1148	ibm-1148	Wolf1148 DOC1 EBCDIC - International
1149	ibm-1149	Ibm1149 IBM EBCDIC - Icelandic (Euro)
1153	ibm-1153	Ibm1153 IBM EBCDIC - 1153 Multilingual/ROECE (Latin-2) with Euro
1155	ibm-1155	Ibm1155 IBM EBCDIC - Turkish (1155 Latin-5 + EURO)
1156	ibm-1156	Ibm1156 IBM EBCDIC - Baltic Multilingual (1156 + Euro)
1164	ibm-1164	Ibm1164 IBM EBCDIC - Vietnamese (1164 + Euro)
1250	windows-1250	Win1250 Windows 1250 (Latin 2)
1251	windows-1251	Win1251 Windows 1251
1252	windows-1252	Win1252 Windows 1252@ISO Latin 1
1253	windows-1253	Win1253 Windows 1253
1254	windows-1254	Win1254 Windows 1254
1255	windows-1255	Win1255 Windows 1255
1256	windows-1256	Win1256 Windows 1256
1257	windows-1257	Win1257 Windows 1257
1258	windows-1258	Win1258 Windows 1258
1361	windows-1361	Win1361 Windows 1361

SCP Number	Lookup table OPS values	Codepage Name
1383	ibm-1383	Ibm1383 IBM 1383 - EUC-CN
5031	ibm-5031	Ibm5031 IBM 5031 - IBM MBCS Simp. Chinese (837+836)
5033	ibm-5033	Ibm5033 IBM 5033 - IBM MBCS Trad. Chinese (835+037)
20105	windows-20105	Win20105 Western European (IA5)
20106	windows-20106	Win20106 German (IA5)
20107	windows-20107	Win20107 Swedish (IA5)
20108	windows-20108	Win20108 Norwegian (IA5)
20866	windows-20866	Win20866 KOI8-R
21866	windows-21866	Win21866 KOI8-U
25524	ibm-25524	Ibm25524 IBM 25524 - IBM MBCS Trad. Chinese (1043+927)
25525	ibm-25525	Ibm25525 IBM 25525 - IBM MBCS Korean (1088+951)
28591	iso-8859-1	Win28591 ISO Latin 1 (iso-8859-1)
28592	iso-8859-2	Win28592 ISO Latin 2 (iso-8859-2)
28593	iso-8859-3	Win28593 ISO Latin 3 (iso-8859-3)
28594	iso-8859-4	Win28594 Baltic (ISO) (ISO Latin)
28595	iso-8859-5	Win28595 Cyrillic (ISO) (iso-8859-5)
28596	iso-8859-6	Win28596 Arabic (ISO) (iso-8859-6)
28597	iso-8859-7	Win28597 Greek (ISO) (iso-8859-7)
28598	iso-8859-8	Win28598 Hebrew (ISO-Visual) (iso-8859-8)

SCP Number	Lookup table OPS values	Codepage Name
28599	iso-8859-9	Win28599 Turkish (ISO) (iso-8859-9)
28605	iso-8859-15	Win28605 Latin 9 (ISO) (iso-8859-15)
38598	iso-8859-8-i	Win38598 Hebrew (ISO-Logical) (iso-8859-8-i)

Notices

Support

Click **here** for full EngageOne Compose documentation and access to your peers and subject matter experts on the Knowledge community.

1700 District Ave Ste 300
Burlington MA 01803-5231
USA

www.precisely.com