ControlCenter PDFsplitter

User Manual

INFRA
LOGIC

www.infralogic.dk

Copyright and Trademarks
ControlCenter PDF splitter User Guide
Version 1.1
Nov 2005
Part number: CCPDFSPLIT 11

http://www.infralogic.dk

InfraLogic ApS
Vinterbuen 12
DK-2750 Ballerup
Denmark
Telephone +45 4464 6531

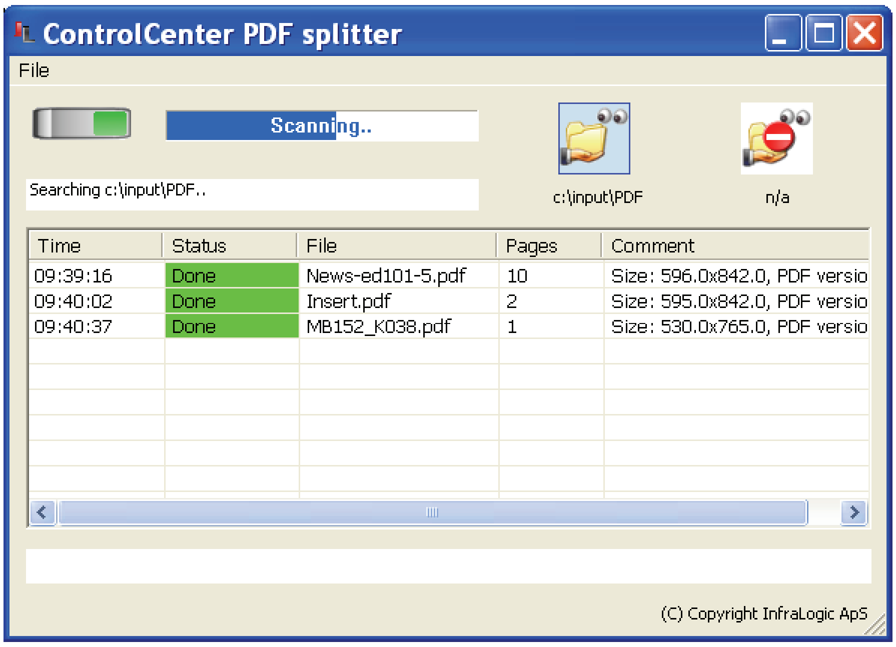
1 Introduction

PDFsplitter is a stand alone Windows based application capable of turning multi-page PDF documents into single PDF pages. Running ControlCenter with PDF input, the PDFs are required to enter as single pages.

PDFsplitter is meant to run unattended watching one or two input folders for PDF documents. Detected documents are split and put in the output folder. An option exists to save the original PDF document in a separate archive folder.

The output is PDF files with the same content and format as the incoming pages. (PDF/X input will generate PDF/X output also).

The filename of output pages will have the original file name appended with the page number.



Searching c:\input\PDF..

Time	Status	File	Pages	Comment
09:39:16	Done	News-ed101-5.pdf	10	Size: 596.0x842.0, PDF versio
09:40:02	Done	Insert.pdf	2	Size: 595.0x842.0, PDF versio
09:40:37	Done	MB152_K038.pdf	1	Size: 530.0x765.0, PDF versio

(C) Copyright InfraLogic ApS

An option exists to offset the appended page number (default start is 1). To offset by a number already in the filename, an input naming convention can be specified so that PDFsplitter will know the offset.

Example (no naming convention):
Input: *XYZ.pdf* Output: *XZY-**1**.pdf, XYZ-**2**.pdf…..*

Example (with naming convention):
Input: *XYZ-6-a.pdf* Output: *XZY-6-a-**6**.pdf, XYZ-6-a-**7**.pdf…..*

2 Configuration

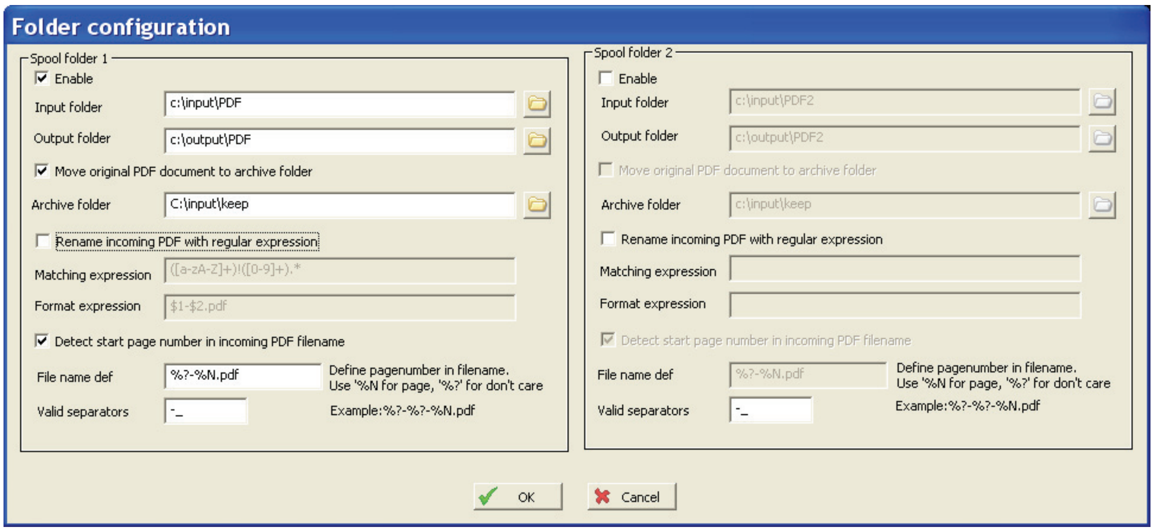
PDFsplitter is easily installed using the install program PDFsplitterSetup.exe found on the installation CD.

The configuration dialogs are accessed via the **File->Configuration..** menu

Folder configuration

The current version of PDFsplitter allows two queues (hot-folders) to be defined. Click the **Enable** checkbox to use a specific hot-folder and enter input folder and output folder name.

Check the archiving option if required. Note that files are deleted from the input folder.



Folder configuration

Spool folder 1
- [✓] Enable
- Input folder: c:\input\PDF
- Output folder: c:\output\PDF
- [✓] Move original PDF document to archive folder
- Archive folder: C:\input\keep
- [ ] Rename incoming PDF with regular expression
- Matching expression: ([a-zA-Z]+)!([0-9]+).*
- Format expression: $1-$2.pdf
- [✓] Detect start page number in incoming PDF filename
- File name def: %?-%N.pdf — Define pagenumber in filename. Use '%N for page, '%?' for don't care
- Valid separators: -_ — Example:%?-%?-%N.pdf

Spool folder 2
- [ ] Enable
- Input folder: c:\input\PDF2
- Output folder: c:\output\PDF2
- [ ] Move original PDF document to archive folder
- Archive folder: c:\input\keep
- [ ] Rename incoming PDF with regular expression
- Matching expression:
- Format expression:
- [✓] Detect start page number in incoming PDF filename
- File name def: %?-%N.pdf — Define pagenumber in filename. Use '%N for page, '%?' for don't care
- Valid separators: -_ — Example:%?-%?-%N.pdf

Rename file using regular expression

PDFSplitter can be used to rename incoming files using so-called regular expressions. Regular expressions are described in appendix A.

Page number detection

In case the output page numbers must be offset by an already defined page number in the file name, check the **Detect start page number..** checkbox. Now the File name scheme must be defined.

The file name scheme must describe the naming convention of the incoming files so that PDFsplitter can find the offset number. Use %N symbol to denote the position of the page number. %? Is used for 'jokers' or wildcards

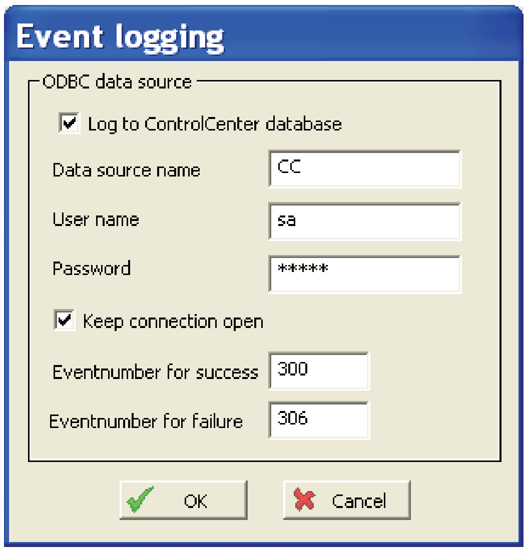
Example: *%?-%N-%j.pdf* will match input file name *SomeName-5-SomeOtherName.pdf*

Database logging

PDFsplitter is prepared to report activity to the ControlCenter system so that errors can be observed from the centralized MonitorCenter application.

To enable logging to ControlCenter, go to menu **File->Logging..**

Key in the ODBC parameters (make sure one is present for the machine running PDFsplitter) and eventcodes to be reported to the ControlCenter database.



Event logging

ODBC data source

[✓] Log to ControlCenter database

Data source name	CC
User name	sa
Password	*****

[✓] Keep connection open

Eventnumber for success	300
Eventnumber for failure	306

Appendix A – Regular expression

The InputCenter naming recognition may use regular expressions for pre-processing of the file names. To utilize the full potential of the very powerful regular expression methodology takes some practice. Recommended reading is *Mastering Regular Expressions by Jeffrey E.F. Friedl (O'Reilly)*.

Make sure to understand the meaning of the terms *match expression* and *format*

expression. Match expressions holds the actual pattern for name recognition. Format expressions are the definitions used for the generation of the renamed file names.

The first part of this appendix sums up the formal syntax of regular expressions. In the last part of the section a number of relevant examples are shown. These examples may serve as relevant starting points for building your own expressions.

A.1 Regular expression syntax

Literals

All characters are literals except: ".", "|", "*", "?", "+", "(", ")", "{", "}", "[", "]", "^", "$" and "\".

These characters are literals when preceded by a "\". A literal is a character that matches itself

Wildcard

The dot character "." matches any single character

Repeats
A repeat is an expression that is repeated an arbitrary number of times. An expression followed by "*" can be repeated any number of times including zero. An expression followed by "+" can be repeated any number of times, but at least once.
An expression followed by "?" may be repeated zero or one times only. When it is necessary to specify the minimum and maximum number of repeats explicitly, the bounds operator "{}" may be used, thus "a{2}" is the letter "a" repeated exactly twice, "a{2,4}" represents the letter "a" repeated between 2 and 4 times, and "a{2,}" represents the letter "a" repeated at least twice with no upper limit. Note that there must be no white-space inside the {}, and there is no upper limit on the values of the lower and upper bounds.
All repeat expressions refer to the shortest possible previous subexpression: a single character; a character set, or a sub-expression grouped with "()" for example.

Examples:
"ba*" will match all of "b", "ba", "baaa" etc.
"ba+" will match "ba" or "baaaa" for example but not "b".
"ba?" will match "b" or "ba".
"ba{2,4}" will match "baa", "baaa" and "baaaa".

Non-greedy repeats
Whenever the "extended" regular expression syntax is in use (the default) then non-greedy repeats are possible by appending a '?' after the repeat; a non-greedy repeat is one which will match the shortest possible string. For example to match html tag pairs one could use something like:
"<\s*tagname[^>]*>(.*?)<\s*/tagname\s*>"
In this case $1 will contain the text between the tag pairs, and will be the shortest possible matching string.

Parenthesis

Parentheses serve two purposes, to group items together into a sub-expression, and to mark what generated the match. For example the expression "(ab)*" would match all of the string "ababab".. In the example the matching engine would contain a pair of iterators denoting the final "ab" of the matching string. It is permissible for sub-expressions to match null strings. If a sub-expression takes no part in a match - for example if it is part of an alternative that is not taken - then both of the iterators that are returned for that sub-expression point to the end of the input string, and the matched parameter for that sub-expression is false. Sub-expressions are indexed from left to right starting from 1, sub-expression 0 is the whole expression.

Non-Marking Parenthesis

Sometimes you need to group sub-expressions with parenthesis, but don't want the parenthesis to spit out another marked sub-expression, in this case a non-marking parenthesis (?:expression) can be used. For example the following expression creates no sub-expressions:
"(?:abc)"*

Forward Lookahead Asserts

There are two forms of these; one for positive forward lookahead asserts, and one for negative lookahead asserts:
"(?=abc)" matches zero characters only if they are followed by the expression "abc". "(?!abc)" matches zero characters only if they are not followed by the expression "abc".

Alternatives

Alternatives occur when the expression can match either one sub-expression or another, each alternative is separated by a "|". Each alternative is the largest possible previous subexpression; this is the opposite behaviour from repetition operators.

Examples:
"a(b|c)" could match "ab" or "ac".
"abc|def" could match "abc" or "def".

Sets

A set is a set of characters that can match any single character that is a member of the set. Sets are delimited by "[" and "]" and can contain literals, character ranges, character classes, collating elements and equivalence classes. Set declarations that start with "^" contain the compliment of the elements that follow.

Examples:
Character literals:
"[abc]" will match either of "a", "b", or "c".
"[^abc] will match any character other than "a", "b", or "c".

Character ranges

"[a-z]" will match any character in the range "a" to "z".

"[^A-Z]" will match any character other than those in the range "A" to "Z".

Note that character ranges are highly locale dependent: they match any character that collates between the endpoints of the range, ranges will only behave according to ASCII rules when the default "C" locale is in effect. For the US localization
model, then [a-z] will match the ASCII characters a-z, and also 'A', 'B' etc, but not 'Z' which collates just after 'z'.

"[[:space:]]" is the set of all whitespace characters.

The available character classes are:

\w in place of [:word:]
\s in place of [:space:]
\d in place of [:digit:]
\l in place of [:lower:]
\u in place of [:upper:]

alnum	Any alpha numeric character.
alpha	Any alphabetical character a-z and A-Z.

Other characters may also be included depending upon the locale.blank Any blank character, either a space or a tab.

cntrl	Any control character.
digit	Any digit 0-9.
graph	Any graphical character.
lower	Any lower case character a-z.
print	Any printable character.
punct	Any punctuation character.
space	Any whitespace character.
upper	Any upper case character A-Z.
xdigit	Any hexadecimal digit character, 0-9, a-f and A-F.
word	Any word character - all alphanumeric characters plus the underscore.
unicode	Any character whose code is greater than 255

Collating elements take the general form [.tagname.] inside a set declaration, where tagname is either a single character, or a name of a collating element, for example [[.a.]] is equivalent to [a], and [[.comma.]] is equivalent to [,]. Multi-character collating elements can result in the set matching more than one character, for example [[.ae.]] would match two characters, but note that [^[.ae.]] would only match one character.
Equivalence classes take the general form [=tagname=] inside a set declaration, where tagname is either a single character, or a name of a collating element, and matches any character that is a member of the same primary equivalence class as the collating element [.tagname.]. An equivalence class is a set of characters that collate the same, a primary equivalence class is a set of characters whose primary sort key are all the same (for example strings are typically collated by character, then by accent, and then by case; the primary sort key then relates to the character, the secondary to the accentation, and the tertiary to the case). If there is no equivalence class corresponding to tagname, then [=tagname=] is exactly the same as [.tagname.].
To include a literal "-" in a set declaration then: make it the first character after the opening "[" or "[^", the endpoint of a range, a collating element.

Line anchors
An anchor is something that matches the null string at the start or end of a line: "^" matches the null string at the start of a line, "$" matches the null string at the end of a line.

Back references
A back reference is a reference to a previous sub-expression that has already been matched, the reference is to what the sub-expression matched, not to the expression itself. A back reference consists of the escape character "\" followed by a digit "1" to "9", "\1" refers to the first subexpression, "\2" to the second etc. For example the expression "(.*)\1" matches any string that is repeated about its mid-point for example "abcabc" or "xyzxyz". A back reference to a sub-expression that did not participate in any match, matches the null string: NB this is different to some other regular expression

matchers. Back references are only available if the expression is compiled with the flag regbase::bk_refs set.

Characters by code
This is an extension to the algorithm that is not available in other libraries, it consists of the escape character followed by the digit "0" followed by the octal character code. For example "\023" represents the character whose octal code is 23. Where ambiguity could occur use parentheses to break the expression up: "\0103" represents the character whose code is 103, "(\010)3 represents the character 10 followed by "3". To match characters by their hexadecimal code, use \x followed by a string of hexadecimal digits, optionally enclosed inside {}, for example \xf0 or \x{aff}, notice the latter example is a Unicode character.

A.2 File name renaming examples using regular expressions

Typically the regular expressions are used to split up a filename into a well defined separated filename. In the following simple examples the match/format expression pair is used to streamline incoming file names. Note the brackets around partial match expressions. These are used to form the output name using the format expression. The first bracket set is mapped to format id $1 etc.

Match expression	Format expression	Incoming filenames	Renamed filenames
[a-zA-Z]*([0-9]+).*	myname-$1	AbC01xyz.tif	myname-01
		Cdcdcdc02	myname-02
.*([0-9]+)[.].*	myothername-$1	abcXyZ123.ext	myothername-123
		xxyy_zz124	myothername-124
([0-9]+).*	anothername-$1	56xYzsD.x.y.z	anothername-56
		57edfr123	anothername-57
([0-9]+)[-_]([0-9]+).*	myname-$1-$2-1	12_98dsdf_a_b_c	myname-12-98-1
		12-99xyz	myname-12-99-1
(.*)[_]([a-zA-Z]+)[_](.*)	$1-$2-$3	x_y_z_01_black_1.tif	x_y_z_01-black-01.tif