RXP Compiler

User Guide

MLNX-15-060476 _v1.5.2 | July 2023

Table of Contents

List of Tables

Chapter 1. Introduction

NVIDIA® RXP™ is a custom-purpose processor developed to efficiently process data to detect matches for a set of user-defined string and regular expression (RegEx) based rules. The RXP Compiler is used to compile RegExes into a format that can be executed by the RXP.

1.1. Scope

This document provides information on the following:

- Pattern syntax supported by the RXP Compiler
- Differences from PCRE behaviour
- Limitations for any constructs
- Pattern optimizations/consolidations
- Best practices for writing good rules

1.2. Related Documentation

The following is a list of related documentation for the content of this document.

- PCRE pattern specification
- PCRE API information
- Mastering Regular Expressions (O'Reilly)

1.3. Typography

The following table describes typographical conventions in NVIDIA documentation.

Table 1. Terms and Definitions

Term	Definition

Job	A unit of data for the RXP to scan. A job can be a packet, packet header, packet payload, packet header and payload, or a block of user-defined data.
Jobset	A set of jobs or packets to be scanned.
RegEx	A common abbreviation for regular expression.
Regular expression	A regular expression is a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A common abbreviation for this is "RegEx".
ROF file	Contains object code used to program the rules memories on the RXP. ROF file names have extension .rof or .rof2, depending on whether it is version 1 or version 2. Unless otherwise stated, the term ROF refers to the latest version, i.e. version 2.
ROFI file	Created by RXP Compiler. Contains the minimum changes required to update the RXP rules memories from one ruleset to another. ROFI file names have extension .rofi or .rof2i, depending on whether it is version 1 or version 2. Unless otherwise stated, the term ROFI refers to the latest version, i.e. version 2.
ROFF file	Contains information required by the RXP Job Generator to create embedded matches for a compiled ruleset.
Ruleset	A list of regular expressions and strings that can be compiled into object code by the RXP Compiler and executed on the RXP.
RXP	High-speed, hardware-accelerated regular expression engine.
Score table	Mechanism used to validate matches detected by RXP.
Thread	A thread refers to a single path through a set of instructions. When a thread reaches a fork type instruction and branches to two instructions, each is referred to as a separate thread. Note that the term thread also still refers to a traditional CPU thread. The context should make it clear as to which thread is being referred to.

1.4. Acronyms

The following table lists the acronyms used in this document.

Table 2. Acronyms

Acronym	Meaning
ASCII	American Standard Code for Information Interchange
API	Application Programming Interface
CSV	Comma Separated Value
ID	IDentifier

PCRE	Perl Compatible Regular Expressions
POSIX	Portable Operating System Interface for uniX
PTPB	Primary Threads Per Byte
RE	Regular Expression
ROF	RXP Object Format (currently at version 2)
ROFI	RXP Object Format Increment (currently at version 2)
ROFF	RXP Object Format Full
RTRU	Run Time Rules Update
RXP	Regular eXpression Processor
SDK	Software Development Kit
SOS	Start Of Subject
UCP	UniCode Properties
XML	eXtensible Markup Language

Chapter 2. RXP Tools Installation

To install NVIDIA® RXP® Tools please use the package manager applicable to the version of Linux you are using.

For Ubuntu, run:

```
sudo dpkg -i <RXP_TOOLS_DEB_FILE>
```

For CentOS, run:

```
sudo rpm -ivh <RXP_TOOLS_RPM_FILE>
```

Chapter 3. RXP Compiler Utility

The RXP Compiler (`rxpc`) is used to compile RegExes into RXP Object Format (ROF) to be executed on the NVIDIA® RXP®. The generated ROF file can be used by a customer application to program the RXP rules memories.

3.1. Usage

```
rxpc [OPTIONS] [ARGS]
```

3.2. Options

The options for the RXP compiler are summarized in the following table.

Table 3. RXP Compiler Options

Short option	Long option	Argument	Type/min/max	Description
-A	--add-rules	FILE	string/-/-	Quick incremental compile add rule feature. Specify a file containing a list of rules to add to a ROF2 file. See section Quick Incremental Compile for more details. Must be used in conjunction with the -I option.
-a	--auto-rule-id		int/1/-	Assign an automatic incrementing ID to each rule.
-B	--pscl	FILE	-/-/-	Specify a prefix selection control list file containing a list of prefixes to be denylisted/graylisted/allowlisted.
-b	--static-denylist	PERCENT	int/1/100	Specify the % of rules that can share a prefix before is denylisted.
-C	--prefix-capacity	CAPACITY	string/-/-	Set the prefix capacity of the RXP. Can be set to 2K, 4K, 8K, 16K or 32K. The default value is 32K.

-c	--checksum	ROFI_FILENAME	string/-/-	Use the end checksum in `ROFI_FILENAME` as the start checksum in the output ROFI file. Can only be used in conjunction with the `--incremental` option.
-D	--divide-ruleset	N	int/1/-	Split the compiled ruleset into N ROF files. Each ROF file will have a subgroup of the rules and all of the ROF files together will make up the complete ruleset.
-E	--force-early-ssid-check		-/-/-	Force the subset ID check to be performed as early as possible. This works best for complex RegEx rules split across many subsets.
-e	--em		-/-/-	Store rules in external memory only.
-F	--force		-/-/-	Force the compilation to complete even if errors are found in the rules.
-f	--file	FILENAME	string/-/-	Specify the rules file to compile.
-g	--virtual-prefix-mode	MODE	int/0/3	Only available from v5.8 onwards. Set the `rxp_virtual_prefix_mode` to use: ▶ 0 (default 0): Only attempt to use virtual prefix if VIRTUAL keyword is specified per rule ▶ 1: Only attempt to use virtual prefix if VIRTUAL keyword is specified per rule or a valid prefix cannot be extracted for that rule ▶ 2: Only attempt to use virtual prefix if VIRTUAL keyword is specified per rule or a prefix cannot be mapped to the rules memories ▶ 3: Only attempt to use virtual prefix if VIRTUAL keyword is specified per rule, or a valid prefix cannot be extracted for that rule, or a prefix cannot be mapped to the rules memories
-h	--help		-/-/-	Display help message and then exit.
-I	--incremental	ROF_FILENAME	string/-/-	Perform an incremental compile using ROF_FILENAME as the base.
-i	--caseless		-/-/-	Switch on global case insensitivity. This option is equivalent to adding the /i modifier to every rule. Caseless mode may still be set and unset within a rule by using the internal (?i) and (?-i) options respectively.
-K	--rule-analysis		-/-/-	Provide files with rules in rank order from most critical to least critical:

				▶ BASEFILENAME_critical_rules_rule_rank.csv ▶ BASEFILENAME_critical_rules_ruleset_rank.csv
-L	--no-locale-support		-/-/-	Turn off locale support.
-M	--max	NUMBER	int_32/1/-	Compile a maximum of NUMBER rules.
-m	--multi		-/-/-	Switch on global multi-line mode. This option is equivalent to adding the /m modifier to every rule. Multi-line mode may still be set and unset within a rule by using the internal (?m) and (?-m) options respectively.
-N	--no-inc-compile-padding		-/-/-	Remove the padding that is normally introduced to assist with incremental compile. This will allow for a more compact rule footprint but is not recommended if you intend to perform an incremental compile with the resulting ROF.
-O	--objective	VALUE	int/1/5	Set compiler objective to prioritize throughput or rules density. Can be set to 1, 2, 3, 4, 5 (default). ▶ Lower value = lower throughput, higher memory density, shorter compile time ▶ Higher value = higher throughput, lower memory density, longer compile time
-o	--output	BASEFILENAME	string/-/-	Write output files to: ▶ BASEFILENAME.rof2 ▶ BASEFILENAME.roff ▶ BASEFILENAME_uncompiled_rules.log ▶ BASEFILENAME_uncompiled_rules_summary.csv ▶ BASEFILENAME_rule_id_lookup_table.csv
-P	--pcre-pre-8-36		-/-/-	Set the space class to not include vertical tab (VT) as was used in PCRE before v8.36.
-p	--ptpb	NUMBER	double/-/-	Set a Primary Threads Per Byte (PTPB) threshold. The default value for this is 0.0001. See Section Automatic Splitting of Rulesets for more information. The PTPB threshold can be switched off by setting it to zero.
-q	--strict-quantifiers		-/-/-	To help with performance, by default the RXP Compiler treats non-fixed bounded quantifiers as unbounded. For example, `.{0,2048}` will be the same as `.*`. This has the caveat of false positives being possible. This switch will ensure that the original construct is used in all

				cases meaning performance will be worse but no false positives will occur.
-R	--max-rep-max	VALUE	int/1/65535	Set the maximum `rep_max` value for all unconstrained repetition constructs. The default for this is 65535. If 1024 is used, `.*` is equivalent to `.{0,1024}`.
-r	--remove-rules	FILE	string/-/-	Quick incremental compile remove rule feature. Specify a file containing a list of rule IDs to remove from a ROF file. See section Quick Incremental Compile for more details. Must be used in conjunction with the `-I` option.
-S	--nosingle		-/-/-	By default, the dot metacharacter matches every character including newline (\x0A). This option switches this off so that dot matches every character except newline. Single line mode may still be set and unset within a rule by using the internal (?s) and (?-s) options respectively.
-s	--enable-split		-/-/-	Enable the automatic alternation splitting.
-t	--threads	NUMBER	int/1/8	Specify the number of threads (1 - 8) to be used during compilation (default is 1).
-U	--utf-16-string		-/-/-	Deal with UTF-16 strings more efficiently. The compiler will assume RegEx constructs like the following are intended to match. ▶ UTF-16 strings: `A\x00?B\x00?C\x00?` With this switch enabled, the compiler will interpret the above RegEx as: `ABCD|A\x00B\x00C\x00`.
-V	--rxp-version	VERSION	string/-/-	The version of the RXP core that the compiled ruleset will be executed on, e.g. '5.7' or '5.8'.
-v	--verbose	LEVEL	int/0/3	Set the verbosity level of terminal output: ▶ 0 - no terminal output ▶ 1 - only warnings and errors ▶ 2 - all output except progress bars ▶ 3 - all output (default)
-W	--disable-bidirectional		-/-/-	This disables the bidirectional rule compilation.
-w	--tpe-max-data-width	WIDTH	int/1/8	Set the maximum TPE data WIDTH in bytes. Values: 1, 2, 3, 4 (default), 5, 6, 7, 8. Experiments have shown that a lower value can be better for rules/data with a shared

				narrow ASCII range and large numbers of prefixes being triggered.
-X	--XML		-/-/-	Switch on XML schema mode. This offers four extra XML related character classes and character class subtraction support.
-x	--free		-/-/-	Switch on free-spacing mode (ignore whitespace in rules).
-Y	--dynamic-denylist-file	FILE	string/-/-	Specify a file to analyze to build up a frequency table of content. This frequency table will be used in conjunction with the % threshold specified in `--dynamic-denylist` to denylist strings based on their frequency, if this exceeds the specified threshold value. This denylist will be written out to a file with the suffix `_generated_pcsl.log`. The prefixes for the ruleset will then be analysed against this and denylisted if they match the frequently occurring strings in the sample data that have been used to create the denylist.
-y	--dynamic-denylist	PERCENT	int/1/100	Specify a % threshold used to denylist strings in the FILE specified using `--dynamic-denylist-file`. This % value represents the % of bytes in the sample data that represent the start of a 1 - 4-byte string. If the string occurs at > the % threshold number of bytes it will then be denylisted.

3.3. Example Usage

A simple example of using the `rxpc` is shown below.

```
# Create a simple rules file, with a single rule "hello\s+world"
echo "1,/hello\s+world/" > ruleset/synthetic.rules

# Compile the rules file. All output files will be prefixed by "rof/synthetic"
rxpc -f ruleset/synthetic.rules -o rof/synthetic
```

The example above creates a ruleset with a single rule containing the alphabet. This is compiled into a ROF file using the `rxpc`. This is created along with the other output files mentioned in section RXP Compiler Output.

When the example above is run the `rxpc` produces output including statistics such as:

```
Info: PPE total virtual prefix usage: 0
Info: PPE total 1-byte prefix usage: 0/256 (0%)
Info: PPE total 2-byte prefix usage: 0/2048 (0%)
Info: PPE total 3-byte prefix usage: 0/2048 (0%)
Info: PPE total 4-byte prefix usage: 1/32768 (0.00305176%)
Info: TPE instruction RAM TCM partition usage: 4096/4096 (100%)
Info: TPE instruction RAM external memory partition usage: 4111/13M (0.0301581%)
Info: TPE class RAM usage: 1/256 (0.390625%)
Info: Estimated threads/byte: 2.592e-10
...
Info: Number of rules compiled = 1/1
```

```
...
Info: Processing ended: Wed Oct 9 10:10:57 2019
Info: Elapsed time: 0 seconds
Info: Compilation rate: 3 rules per second
```

The first eight lines show the memory usage for prefixes, instructions, and classes. Next the PTPB value (see section Incremental Compile) attributed to the entire ruleset followed by the number of rules that have successfully compiled. Finally, some compilation statistics are displayed. For the compilation rate to be accurate a larger ruleset needs to be compiled.

3.4. Input

Input	Link
Rules file (.rules)	See section Input Rules Files for file format.

3.5. Output

Output	Link
Rule ID lookup table (.csv)	See section Rule ID Lookup Table File for file format.
RXP object format (.rof2)	See section Remove Rules File for file format.
RXP object format increment (.rof2i)	See section ROF File for file format.
RXP object format full (.roff)	See section ROFI File for file format.
Uncompiled rules log (.log)	See section Uncompiled Rules File for file format.
Uncompiled rules summary (.csv)	See section Uncompiled Rules File for file format.
Critical rules rank (.csv)	See section Critical Rules Rank File for file format.
Generated_pscl (.log)	See section Prefix Selection Control List Files for file format.
Rule_direction_analysis (.csv)	See section Rule Direction Analysis File for file format

3.6. Error Reporting

Any errors that are encountered during compilation will be displayed to the terminal and processing will halt. If the force (`-F`) option is specified, processing will continue until all rules have been processed. The error details for any rules that could not be compiled when force (`-F`) is used will be written to a log file (see section Critical Rules Rank File). However, should the subset ID be found to be out of range, then processing will halt regardless of whether `-F` is specified or not.

The RXP compiler logs four distinct type of errors during the compilation process. These are described in the following sections.

3.6.1. Rules File Format Error

This type of error points to an error in the rules file format. This can be corrected to allow compilation. A file name, line number and pointer to problem will be provided to aid in debugging. See the following table for a full list of error strings and descriptions.

Table 4. Rules File Format Errors Reported by RXP Compiler

Error string	Description
no rule found	There was no rule found where it was expected to be in the input rules file.
no subset_rule_id found	There was no subset rule_id found where it was expected to be.
unrecognized or duplicated modifier	There is an issue with the modifiers in a rule. They may be unrecognized or duplicated.
file could not be opened	There was an issue opening the specified file.
unrecognized line format	This specific line was not recognized.
subset_id out of range	This indicates that a subset ID specified falls outside the range 1-65535. If this is detected processing will halt, even when using the -F switch.
subset_rule_id out of range	This indicates that a subset rule ID specified falls outside the range 0-4,294,967,295.

3.6.2. Not Enough Resources

This type of error means that the current version of the RXP does not have enough resources to execute the rule. See the following table for a full list of error strings and descriptions.

Table 5. "Not Enough Resource" Errors Reported by RXP Compiler

Error string	Description
class resources exceeded	There is no more room for any new RegEx classes
exceeded maximum PTPB threshold	See section Incremental Compile for more details
insufficient prefix resources	There is not enough prefix memory to store all the rules
maximum number of back references possible is 16	There are more than 16 back references used in a rule

maximum primary thread count exceeded	The rule causes too many primary threads to be spawned
maximum secondary thread count exceeded	The rule causes too many secondary threads to be spawned
nested repetition prefixes not yet supported	It is not possible to extract a prefix from a rule with nested repetition at the start (e.g. `/(A*A*A*A+)+/`)
no valid prefixes found	No valid prefixes can be found in the rule (e.g. `/.*/`)
not enough resources available	There are not enough resources available on the RXP to execute the rule
prefix overflow	A prefix could not be extracted from the rule due to excessive alternation paths

3.6.3. Syntax Error

This type of error points to an error in the rule itself. This can be corrected to allow compilation. A file name, line number and pointer to problem will be provided to aid in debugging. See the following table for a full list of error strings and descriptions.

Table 6. Syntax Errors Reported by the RXP Compiler

Error string	Description
'/' character must be escaped	The '/' character must be escaped with '\' unless it is the beginning and last character in a rule
ASCII control character must be an alphabetic character	See RXP Pattern Syntax and RegEx Writers Guide for more details
cannot peek as at end of data stream	Logic of RegEx elements is not correct. That is, only part of a RegEx construct specified at end of rule.
cannot peek back as at beginning of data stream	Logic of RegEx elements is not correct. That is, only part of a RegEx construct specified at beginning of rule.
cannot unget as at beginning of data stream	Logic of RegEx elements is not correct. That is, only part of a RegEx construct specified at beginning of rule.
class subtraction must always be the last element in its containing character class	See RXP Pattern Syntax and RegEx Writers Guide for more details
found '^' character in middle of rule	The '^' character should only occur at the beginning of the rule (unless escaped with '\')
found '$' character in middle of rule	The '$' character should only occur at the end of the rule (unless escaped with '\')
invalid capturing group	See RXP Pattern Syntax and RegEx Writers Guide for more details

invalid posix character class definition	See RXP Pattern Syntax and RegEx Writers Guide for more details
no functional constructs found in rule	The rule does not contain any constructs and is essentially null
numbered reference exceeds 255	The maximum value allowed for any numbered references is 255
out of order range in character class	In a class range the max value occurs before the min value (e.g. `/[d-a]/`)
out of order repetition quantifiers	In a repetition, the max value is less than the min value (e.g. `/A{4,0}/`)
reference in conditional statement to non-existent capture number	Capture that is referenced in conditional statement does not exist
reference in conditional statement to non-existent named capture	Capture that is referenced in conditional statement does not exist
reference to non-existent capture number	Capture that is referenced does not exist
reference to non-existent named capture	Capture that is referenced does not exist.
relative reference exceeds the number of captures at its point	See RXP Pattern Syntax and RegEx Writers Guide for more details
repetition quantifier exceeds the maximum repetition value	The maximum repetition value is 4096
unclosed group, character pointer has exceeded the rule length	The end of the rule has been reached and an open group has not been closed
unclosed parenthesis	The end of the rule has been reached and an opening parenthesis has not been closed
unexpected character	The character encountered was not expected to occur in/after the current RegEx construct
unmatched parenthesis	A closing parenthesis has been encountered without a corresponding opening parenthesis
unterminated posix character class definition	The end of the rule has been reached and a POSIX character class has not been closed

3.6.4. Unsupported Feature

This type of error means that the current version of the RXP does not support a specific RegEx feature. See RXP Pattern Syntax and RegEx Writers Guide for more details.

3.7. Incremental Compile

The RXP compiler allows a rules file to be compiled incrementally using an existing ROF file as a baseline. This process will create a ROF Increment (ROFI) file which contains only the differences between the baseline ROF file and the new ROF file. This is used to perform a Run-Time Rules Update (RTRU) which means the RXP rules memories can be updated whilst in operation. There are two incremental compile modes available; quick and normal which are described below in sections Quick Incremental Compile and Normal Incremental Compile respectively.

3.7.1. Quick Incremental Compile

This feature allows rules to be quickly added to and quickly removed from a ROF file. As quick adds/removes occur the ruleset performance will gradually get worse and the rules memory usage will lose optimal compression. This is because the rules are added to the pre-existing data-structures and do not benefit from much of the rule optimization algorithms. The number of rules that can be added to a ROF file is also limited by the buffers present in the pre-existing data-structures. Due to the caveats associated with the quick incremental compile, it is required to run a full recompile of the ruleset to clean up if any performance issues are observed.

The main benefit of quick incremental compile is that the rules can be added and removed extremely fast.

The main disadvantage is that as more rules are added and removed the ruleset performance will suffer.

For the quick remove rules feature the file format can be found in section Remove Rules File. The following command can be used to quick remove rules:

```
rxpc --incremental ROF_FILENAME --remove-rules LIST_OF_RULE_IDS_FILENAME
```

Or:

```
rxpc -I ROF_FILENAME -r LIST_OF_RULE_IDS_FILENAME
```

For the quick add rules feature the file format can be found in section Input Rules Files. The following command can be used to quick add rules:

```
rxpc --incremental ROF_FILENAME --add-rules LIST_OF_RULES_FILENAME
```

Or:

```
rxpc -I ROF_FILENAME -e LIST_OF_RULES_FILENAME
```

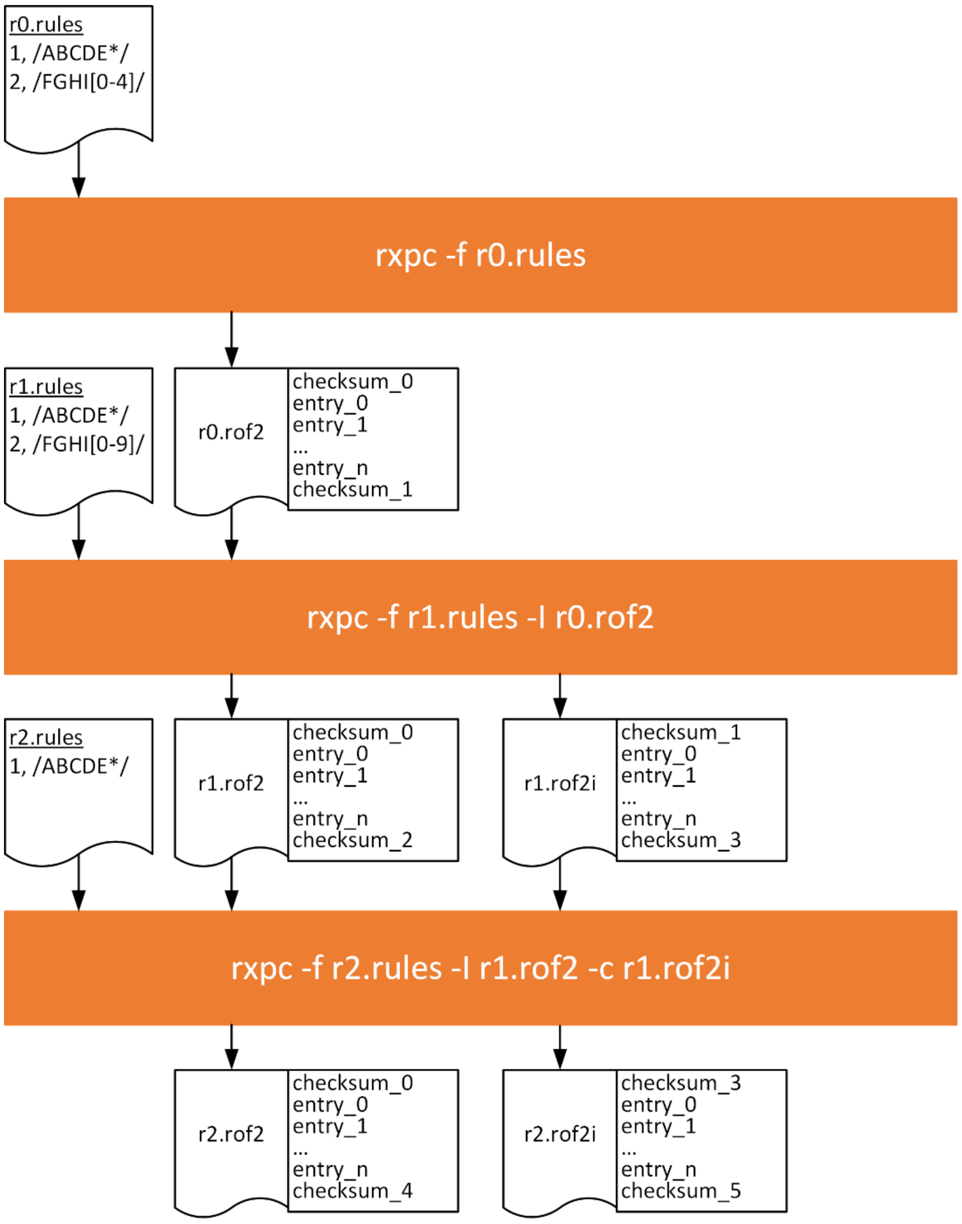
3.7.2. Normal Incremental Compile

The normal incremental compile will take longer than a standard rules compilation. It will however avail of all the rule optimization algorithms. After performing a RTRU with the resulting ROFI the rules memories will be as optimal as they would be from a full compilation.

- The main benefit of normal incremental compile is that the rules memories will be optimal.

- The main disadvantage from normal incremental compile is that it takes a long time to compile the rules.

The usage flow for a two-revision normal incremental compile is illustrated in the following figure.



In this figure, `r0.rules` is the original ruleset or revision zero. This is compiled in the normal way resulting in `r0.rof`. The "[0-4]" class in rule ID two is then changed to "[0-9]" resulting in `r1.rules` (revision one). To incrementally update the RXPs rules memories for each new revision the incremental compile feature can be used. It can be seen that `r1.rof2i` is the result of incrementally compiling `r1.rules` using `r0.rof2` as a baseline.

Checksums are used to validate the integrity of the ROF data when programmed into the rules memory. The checksum at the beginning of `r1.rof2i` is the same as the checksum at the end of `r0.rof2`. This is checked at the beginning of the rules programming sequence to ensure the RXP rules memories are configured with `r0.rof2` before they can be updated with `r1.rof2i`. If the checksum does not match, then the rules programming sequence will not complete. In this case the RXP must be reinitialized and then `r1.rof` can be used to configure the rules memories from scratch.

Finally, `r2.rules` (revision two) has been created by removing rule ID two from `r1.rules`. It can be seen that `r2.rof2i` is the result of incrementally compiling `r2.rules` using `r1.rof2` as a baseline. If the rules memories have been successfully updated using `r1.rof2i` previously, then `r1.rof2i` must also be provided using the `-c` option so the end checksum from `r1.rofi` can be used as the check at the beginning of the rules programming sequence for `r2.rof2i`. If the rules had been programmed from scratch with `r1.rof2` then the `-c` option would not be required. In this case, the end checksums from `r1.rof2` would be used as the check at the beginning of the rules programming sequence for `r2.rof2i`. There is currently no limit to the number of revisions in this sequence.

3.8. Automatic Splitting of Rulesets

The `-D` or `--divide-ruleset` option allows the ruleset to be split across N ROF files. Each of the resultant ROF files will contain a subgroup of the rules and all the ROFs together will make up the complete ruleset. When this option is used, the ruleset is analyzed to provide the most even spread of rules between the ROF files, the rules are then compiled according to the results of this analysis. The resultant ROF files will take the following format, considering in this case that the `BASEFILENAME` in `-o` was set to `rof/synthetic`:

```
./rof/synthetic-0.rof2
./rof/synthetic-0.roff
./rof/synthetic-1.rof2
./rof/synthetic-1.roff
.
.
.
./rof/synthetic-N.rof2
./rof/synthetic-N.rof2
```

3.9. Automatic Rules Normalization

The `-s` or `--enable-split` option triggers the split alternation functionality. This can offer benefits such as better resource sharing between rules and in some cases better performance. If a rule contains one or more alternations, a rule will be generated internally for each branch of the alternation/s, these rules use the same `rule_id` as the parent so this is transparent to the application. The split alternation functionality can be enabled in the following ways:

- Globally with `-s` or `--enable-split` option. In this case, the RXP Compiler will determine the optimal way to split a rule.
- It can be enabled per rule with the `o` modifier, or explicitly disabled with `O` modifier.

In any case, a rule alternation will not be split if it generates more than 10240 rules. In some cases, the split alternation functionality can cause extra matches over PCRE to occur. It can also have a negative impact on performance in some extreme cases.

3.10. Automatic Prefix Denylist Generation

There are two modes for automatic prefix denylist generation: static, and dynamic. These modes can be run simultaneously if required and both output a Prefix Selection Control List (PSCL) file. They are explained in the sections below.

3.10.1. Static

This uses the `-b` or `--static-denylist` option along with a percentage value. The percentage value is used to indicate the percentage of rules that can share a prefix before it is denylisted. The percentage value can be tuned according to requirements. If set too low, then the resultant PSCL may not be very effective. If set too high, then the resultant PSCL may not allow any scope at all for prefixes in the ruleset.

3.10.2. Dynamic

This allows a data file to be specified using `-Y` or `--dynamic-denylist-file`. This file is then analyzed to build up a frequency table of content. This frequency table is then used in conjunction with the percentage threshold specified using `-y` or `--dynamic-denylist`. The percentage value is used to indicate the percentage of byte positions in the file at which a string occurs before it is denylisted. The percentage value can be tuned according to requirements. If set too low, then the resultant PSCL may not be very effective. If set too high, then the resultant PSCL may not allow any scope at all for prefixes in the ruleset. The resulting PSCL from the data file is then applied to the list of prefix candidates for the ruleset.

3.11. Primary Threads Per Byte (PTPB)

The default value for this is 0.0001. When a prefix is detected, one or more primary threads can be triggered. Each primary thread is dispatched to a Thread Engine (TE). The TE executes instructions associated with the search for a full match for one or more rules. A primary thread can trigger zero or more secondary threads during execution. These secondary threads are also managed by the TE. The RXP has a finite number of primary threads that can be executed at any one time. Due to this, any rules performance can be characterized for a set of data by the number of primary threads it generates for every byte. To use for an indicator as to how well a rule will perform the RXP Compiler calculates its PTPB value based on random data. As the PTPB value increases, performance decreases. One rule with a high PTPB value can have a severe effect on the entire ruleset. The PTPB value is calculated for a given rule using the following equations:

Equation 1:

$$\boldsymbol{U = \frac{256}{230}}$$

Equation 2:

$$\boldsymbol{A = \frac{1}{64}}$$

Equation 3:

$$\boldsymbol{D_1 = \frac{1}{2^{8*1}}}$$

Equation 4:

$$\boldsymbol{D_2 = \frac{1}{2^{8*2}}}$$

Equation 5:

$$\boldsymbol{D_4 = \frac{1}{2^{8*4}}}$$

Equation 6:

$$A_{1a} = D_1 * U * A$$

Equation 7:

$$A_{2a} = D_2 * U * A$$

Equation 8:

$$A_{4a} = D_4 * U * A$$

Equation 9:

$$A_{1u} = D_1 * U$$

Equation 10:

$$A_{2u} = D_2 * U$$

Equation 11:

$$A_{4u} = D_4 * U$$

Equation 12:

$$P_a = (A_{1a} * N_{1a} * F_{1a}) + (A_{2a} * N_{2a} * F_{2a}) + (A_{4a} * N_{4a} * F_{4a})$$

Equation 13:

$$P_u = (A_{1u} * N_{1u} * F_{1u}) + (A_{2u} * N_{2u} * F_{2u}) + (A_{4u} * N_{4u} * F_{4u})$$

Equation 14:

$$P = P_a + P_u$$

Where:

A	Anchored adjuster (considers a 64 byte job)
A1a	Adjusted probability per one-byte anchored prefix
A2a	Adjusted probability per two-byte anchored prefix
A4a	Adjusted probability per four-byte anchored prefix
A1u	Adjusted probability per one-byte unanchored prefix
A2u	Adjusted probability per two-byte unanchored prefix
A4u	Adjusted probability per four-byte unanchored prefix
F1a	Average primary threads per one-byte anchored prefix
F2a	Average primary threads per two-byte anchored prefix
F4a	Average primary threads per four-byte anchored prefix
F1u	Average primary threads per one-byte unanchored prefix
F2u	Average primary threads per two-byte unanchored prefix
F4u	Average primary threads per four-byte unanchored prefix
D1	Discrete uniform distribution function per one-byte prefix
D2	Discrete uniform distribution function per two-byte prefix
D4	Discrete uniform distribution function per four-byte prefix
N1a	Number of one-byte anchored prefixes
N2a	Number of two-byte anchored prefixes
N4a	Number of four-byte anchored prefixes
N1u	Number of one-byte unanchored prefixes
N2u	Number of two-byte unanchored prefixes
N4u	Number of four-byte unanchored prefixes
P	Primary threads per byte
Pa	Primary threads per byte for all anchored prefixes

Pu	Primary threads per byte for all unanchored prefixes
U	Uppercase adjuster

The value P calculated from the equations above represents the PTPB value for the rule. So if a rule for example had one unanchored one-byte prefix like the rule /AB*CDEF/ this would give a PTPB value of 0.004 or 1/230. This means that in random data this rule will trigger one primary thread for every 230 bytes of data. If a rule has one anchored one-byte prefix like the rule /^AB*CDEF/ this would give a PTPB value of 0.000068 or 1/14,720. This means that in random data this rule will trigger one primary thread for every 14,720 bytes of data. If a rule has one unanchored two-byte prefix like the rule /ABC*DEF/ this would give a PTPB value of 0.000017 or 1/58,880. This means that in random data this rule will trigger one primary thread for every 58,880 bytes of data.

It is important to note that the PTPB value is a theoretical indicator calculated on uniformly distributed random data to give an idea of rule performance. Internet traffic for example has more of a normal distribution so will not have an identical PTPB value for a given rule. The equations could be modified to suit the characteristics of the target data if required.

Chapter 4. Data Flow and File Formats

The following table provides a brief description of all the file formats used/produced by the RXP Compiler and RXP evaluation utilities.

Table 7. File Formats Used by RXP Tools

Filename	Output from	Input to	Description
`*.rules`	--	`rxpc`	The list of rules to be compiled. See section Input Rules Files.
`*.denylist`	--	`rxpc`	A file containing a list of newline separated strings that are not desirable to use as prefixes. See section Prefix Selection Control List Files.
`*_generated_pscl.log`	`rxpc`	--	A file containing a list of newline separated strings that are not desirable to use as prefixes. Generated when static or dynamic denylist options are used. See section Prefix Selection Control List Files.
`*_rule_id_lookup_table.csv`	`rxpc`	`rxpe`	A file to match the automatically generated rxp_rule_id with user-defined subset_rule_id. See section Rule ID Lookup Table File.
`*_uncompiled_rules.log` `*_uncompiled_rules_summary.csv`	`rxpc`	--	Log file containing details of errors found during the compilation process. See section Uncompiled Rules File.

`*_critical_rules_rule_rank.csv`	`rxpc`	--	Lists the rules ranked based on their PTPB value from worst to best. See section Critical Rules Rank File.
`*.rule_id`	`rxpc`	--	List the rule IDs of the rules that should be removed from a ROF file. See section Remove Rules File.
`*.rof2`	`rxpc`	`rxpc`, `hra`	The RXP object format file that contains the object code used to configure the RXP. See section ROF File.
`*.rof2i`	`rxpc`	`rxpc`, `hra`	The RXP object format increment file that contains the minimal number of entries required to configure the RXP. See section ROFI File.
`*.roff`	`rxpc`	`rxpj`	The RXP object format full file that is used to embed matches within a jobset. See section ROFF File.

4.1. CSV File Format

Many of the files generated and used by the RXP Compiler and RXP evaluation utilities use Comma-Separated Values (CSV) format files (although may use a different file extension). CSV files can be viewed and edited in widely available spreadsheet applications.

A CSV file stores tabular data (numbers and text) in a plain-text format. It consists of any number of records, separated by line breaks of some kind; each record consists of fields, separated by a literal comma. Usually, all records have an identical sequence of fields.

Zero or more whitespace characters can be placed between the field values and the commas e.g:

```
72, 2, 0, 512
1,2     ,3, 4
```

Comments must be placed on a separate line and start with a '#'.

4.2. Input Rules Files

Rules files include certain keywords used to define how the rules will be interpreted by the `rxpc` utility, as described in the following table.

Table 8. Supported Rules File Keywords

Keyword	Range	Description
`direction`	U/D/UD/DU	Only available from v5.8 onwards. Optional column. Specify the chosen direction for a rule. ▶ U – these prefixes can only be chosen in the first eight bytes of a rule if the rule is longer than eight bytes. ▶ D – these prefixes can only be chosen in the last eight bytes of a rule if the rule is longer than eight bytes. ▶ UD – these prefixes can only be chosen in between the first and last eight bytes of a rule if the rule is longer than 16 bytes. ▶ DU – these prefixes can only be chosen in between the first and last eight bytes of a rule if the rule is longer than 16 bytes.
`prefix`		Optional column. Specify the chosen prefixes for a rule. Bytes can be represented in \x notation, and strings are case-insensitive. Specified prefixes can be between one and four bytes. Multiple prefixes should be comma separated and contained in braces. The keyword VIRTUAL can be used if a virtual prefix is required for the rule.
`rule_id`	1-4,294,967,295	The rule identifier that can be used as a cross reference for the rule. This must be specified just before the rule.
`subset_id`	1-65,535	Defines rules subset identifier. If no subset identifier is specified, the default value of one will be used.
`@`		Indicates a label to be attached to the next rule.
`#`		Indicates a comment line.

Each rule subset is delineated using a rules subset identifier number, `subset_id`. For example:

```
# rules subset to detect some simple patterns
subset_id = 1
  # format for each rule is: [subset_rule_id], rule
  # subset_rule_id values are local to each subset
  1, /ABCDEFGH/
  2, /HELLO\s+WORLD/

  prefix=ABCD, 3, /ABCD1234/
  prefix={ABCD,1234}, rule_id=4, /ABCD|1234/

# another subset
subset_id = 7
  1, /XYZ/
  @ attach this text label to following rule for use in application
  2, /AAAA.*BBBB/
```

For details on the RXP RegEx support please see section Regular Expression Support.

Once the rules are compiled, if the `--incremental` option is specified, each is assigned a unique incremental rxp_rule_id (see section RXP Compiler Options). This is the value that will be returned with each match that can then be mapped to the original rule using the `rule_id_lookup_table.csv` file (see section Rule ID Lookup Table File).

If no subset identifier is specified in a rules file, then the default value of one will be used. The subset identifier of zero is reserved and should not be used.

This file format is also used for the quick incremental compile add rules function. Please see Quick Incremental Compile for more details.

4.3. Prefix Selection Control List Files

Prefix selection control list files are used to specify strings that contain values that should be denylisted/graylisted/allowlisted when the RXP Compiler is selecting a prefix. They also support comments as described in the following table.

Table 9. Supported Prefix Denylist File Keywords

Keyword	Description
`#Denylist`	The RXP Compiler will not use any prefixes that are contained in strings on lines following this header. If there are no options but to use a denylisted prefix, the rule will not be compiled.
`#Graylist`	The RXP Compiler will try not to use any prefixes that are contained in strings on lines following this header. It will settle for these if there are no other options.
`#Allowlist`	The RXP Compiler will try to use any prefixes that are contained in strings on lines following this header. If it cannot, it will use other prefixes.
`#`	Indicates a comment line.

Each prefix selection control list string is delineated by a newline, bytes can be represented in `\x` notation, and strings are case-insensitive. For example:

```
#Denylist
# for searching Internet traffic something like this
http
www
.com

# Or the same three strings in \x hexadecimal notation
\x68\x74\x74\x70
\x77\x77\x77
\x2e\x63\x6f\x6d

# or they can all be specified using one line
httpwww.com

#Graylist
# maybe common IP addresses
192.168.1.1
192.168.0.1

#Allowlist
UNCOMMON_STRING

#Denylist
```

```
.co.uk
.org
.gov
```

The content of each string following the `#Denylist` and `#Graylist` header will be avoided when choosing a prefix meaning that other prefixes will have preference. These should be strings whose content is known to have a high incidence in the target data. For example, for rules that will be used to scan Internet packets http, www and .com may be used (as above). If there is no other option other than the denylisted prefix, then the rule will not be compiled. If there is no other option other than the graylisted prefix, then it will be used as a last resort.

The content of each string following the `#Allowlist` header will have preference over any other prefix.

4.4. Rule ID Lookup Table File

The `<variant>_rule_id_lookup_table.csv` file provides a means for the user/application to map the rxp_rule_id from the returned match to the `subset_rule_id` in the input rules file. This file is comma-separated and can be viewed in a spreadsheet application. Each entry consists of four comma-separated fields: `rxp_rule_id`, `subset_rule_id`, `subset_id` and `rule`.

Table 10. Rule ID Lookup Table File Fields

Match Field	Range	Description
`subset_id`	RXP v5.7 – 1-4,095 RXP v5.8 – 1-65,535	Subset identifier
`rule_id`	1-4,294,967,295	User-specified rule identifier
`rxp_rule_id`	1-4,294,967,295	Incremental rule identifier when `--auto-rule-id` is used. Otherwise this will always be the same as rule_id.
`rule_direction`	RXP_RULE_DIRECTION_TYPE	Rule direction, only applicable from RXP v5.8 onward
`rule`		The actual rule
`#`		Indicates a comment line

Each entry is placed on a separate line. For example, the resulting file for the input rules given in section Input Rules Files would be:

```
# use comma separated fields in the following order:
# subset_id, rule_id, rxp_rule_id, rule_direction, rule
1, 1, 1, 0, /ABCDEFGH/
2, 2, 1, 0, /HELLO\s+WORLD/
3, 1, 7, 0, /XYZ/
4, 2, 7, 0, /AAAA.*BBBB/
```

4.5. Rule Direction Analysis File

The `<variant>_rule_direction_analysis.csv` lists each prefix chosen for each rule along with its direction.

Table 11. Rule Direction Analysis File Fields

Match Field	Range	Description
`rule_id`	1-4,294,967,295	Rule identifier
`prefix`		User-specified rule identifier within subset
`dir`	U|D|UD|DU	This indicates the direction the job is processed on matching the prefix: ▶ U = Walk up ▶ D = Walk down ▶ UD = Walk up then walk down ▶ DU = Walk down then walk up
`rule`		The actual rule
`#`		Indicates a comment line

At the end of the file a summary is given to indicate the number of each walk direction chosen for the ruleset. For example, the resulting file for the input rules given in section Input Rules Files would be:

```
# rule_id, prefix, dir, rule
1, ABCD, U, ABCDEFGH
2, HELL, U, HELLO\s+WORLD
3, XYZ, U, XYZ
4, AAAA, U, AAAA.*BBBB

#Walk direction summary
:4
D :0
UD:0
DU:0
```

4.6. Uncompiled Rules File

If the `-F` or `--force` option is used in the RXP Compiler, it will carry on and compile as many rules as it can. Any errors encountered in rules will be written to a log file. Each line in the log file will give information on the error encountered and other information that may assist in getting the problem rule to compile. There is also an uncompiled rules summary file that lists a count of each individual error encountered. It is important to note that these files will only be created if errors are found in the target ruleset. For more information on the types of errors and error reporting in the RXP Compiler see section Example Usage.

4.7. Critical Rules Rank File

This file ranks the rules based on their PTPB value (see section Incremental Compile) from worst to best. This file can be used to identify the worst performing rules and remedial action can be taken if necessary.

4.8. Remove Rules File

This file is used for the quick incremental compile remove rule feature. For more details see sections RXP Compiler Options and Quick Incremental Compile.

```
# This is an example of a valid remove ruleset file
# This will remove rules 1 and 3
1, 3

# And 10 as well
10
```

4.9. ROF File

A ROF file consists of one or more lines of comma-separated entries to:

1. Define the memory contents of the RXP's rules memories.
2. Expected register values for sanity checking the RXP's CSRs and the RXP's RTRU_CSR CHECKSUM registers once memory contents have been programmed into the RXP's rules memories.

Each line of the ROF file consists of three comma-separated fields:

1. Type: Indicates the type of the ROF entry, see Table 56 for more information.
2. Address: 32-bit address value prepended with 0x|0X
3. Data: 64-bit data value prepended with 0x|0X

The lower 24-bits of the address field shall be used, while the upper 8-bit shall be set to 0x00.

Table 12. ROF Entry Types

ID	Name	Description
0	`legacy_instruction`	The legacy ROF instruction type, kept to maintain a degree of backwards compatibility.
1	`check_csr_eq`	Read the specified CSR and check that it is equal to the data.

2	`check_csr_gte`	Read the specified CSR and check that it is greater than or equal to the data.
3	`check_csr_lte`	Read the specified CSR check that it is less than or equal to the data.
4	`check_csr_checksum`	Read the specified checksum CSR and check that it is equal to the data.
5	`check_csr_checksum_excluding_em`	Read the specified checksum CSR and check that it is equal to the data. This is used in a system that shares the external memory with the RXP.
6	`im`	Represents a write to the RXP internal memory.
7	`em`	Represents a write to the RXP external memory

Comments must be placed on a separate line and start with a '#'.

An example ROF file is shown below:

```
#type, address, data
# initial checksums, the RTRU_CSR address should be provided in addr field
4, 0x00010010, 0x0000000000000000
4, 0x00010011, 0x0000000000000000
4, 0x00010012, 0x0000000000000000

# rules memory entries
7, 0x00123456, 0x0706050403020100
7, 0x00123457, 0x0706050403020100
…
# final checksums, the RTRU_CSR address should be provided in addr field
4, 0x00010010, 0x0000000012345678
4, 0x00010011, 0x0000000023456789
4, 0x00010012, 0x000000003456789A
```

4.10. ROFI File

A ROFI file has the same structure as a ROF file. It only contains the differences between an old and new ROF file. It can be used to update the RXP rules memories to a new ROF file requiring minimal changes.

4.11. ROFF File

A ROFF file is a binary file used by the `rxpj` to embed matches within a jobset.

Chapter 5. Regular Expression Support

The RegEx standard supported by the RXP Compiler is based on the PCRE pattern specification (see section Related Documentation). The RXP Compiler supports a subset of this standard with the supported and unsupported constructs summarized in sections Backslash and Word Boundary respectively. The RXP Compiler also supports some functionality in addition to the PCRE specification. The extra functionality bolts on to the top of the PCRE support and is summarized in the following table.

Table 13. Additional Functionality Supported by the RXP Compiler

Feature	Description
Anchored to offset	This allows the traditional Start of Subject (SOS) anchor to be extended in order to allow the start point to be redefined as a user-specified value (see section Anchored to Offset).
XML schema	The XML schema features are not enabled by default. In order to use them they must be switched on. The XML schema offers four extra character classes (see table "Supported XML Schema Classes" in section Predefined Classes) and the character class subtraction feature (see table "Supported Character Class Notation" in section User-defined Character Classes). It is important to note that when the XML schema is enabled, the \c escape sequence no longer applies to a control character (see section Non-printing Characters) but now the XML class as shown in table "Supported XML Schema Classes" in section Predefined Classes.

5.1. Assumptions

The RXP compiler makes the following assumptions:

1. All rules are ungreedy. This means the RXP will always report back the shortest possible match. For example, for the RegEx `/AB*/` and the data `ABBBBB` the RXP would match A as this is the shortest match. If this was scanned by PCRE in greedy mode it would match `ABBBBB` as this is the longest match.
2. To save hardware resources, all parentheses are non-capturing by default (they will only capture if back referencing is required).

5.2. Backslash

The "\" (backslash) metacharacter is used in many of the constructs listed below. In all of its usage scenarios, the backslash metacharacter will enforce a different meaning on the character directly following it. There is a set of metacharacters that can follow a backslash metacharacter and these metacharacters also determine what function that will be performed by the resulting "backslash+character" metasequence. The following is a list of all the backslash metacharacters that the RXP compiler supports:

- Quoting. For example, `/\*\?/` will match literally `*?`.
- Non-printing characters. For example, `/\n/` will match the newline character.
- Hexadecimal formats. For example, `/\x0A/` will match the newline character.
- Octal formats. For example, `/\012/` will match the newline character.
- Predefined general classes. For example, `/\s/` will match any whitespace character.
- Back references. For example, `/(ABC)\1/` will match `ABCABC`.
- Simple assertions. For example, `/\AABC\Z/` will match `ABC` with the `A` occurring at the beginning of the subject and `C` occurring at the end of the subject.

5.3. Supported Constructs

The following table provides a brief description of each RegEx construct the RXP Compiler supports. These are described in more detail in the sections that follow.

Table 14. Regex Constructs Supported by the RXP Compiler

Construct	Example	Description
Alternation	/ABC|DEF/	Functions in the same manner as the logical OR. The example will match ABC or DEF.
Anchored to Offset	/^.{12}ABCD$/	Match must occur at the specified offset to the beginning of the subject. In the example, ABCD must begin at byte pointer 12.
Anchoring	/^ABCD$/	Anchors mean the match must occur at the beginning or end of the data stream. Anchors can also apply to \n characters if the m modifier is selected (see section Modifiers). The example will only match ABCD if this is the case.
Back References	/<(A)>BCDE<\1>/	Back references are variables that refer to text matched earlier in the RegEx within capturing parentheses. In the example \1 will match A as it has been captured within the parentheses.

Capturing Parentheses	/(ABCD)|(DEFG)\1/	For back referencing it is required to capture a match to be referenced. This is achieved through capturing parentheses. In the example, ABCD will be captured and stored in \1 whereas DEFG will not be captured unless a \2 back reference is specified.
Conditional Statement	/1234(A)?(?(1)B|C)/	A special type of back reference that allows an if/then/else statement to be expressed in an RegEx in the form: (?if then | else). In the example B will only be matched if an A had matched before. Otherwise C will be matched.
Dot Metacharacter	/ABCD.F/	Can match any ASCII value except newline. The example will match ABCD followed by any character except newline then followed by F.
Hexadecimal Formats	/\xFF\xFF/	Use the \x delimiter to signify hexadecimal values. The example will match 2 bytes in a row each set to 255.
Inline Comments	/ABC(?#DEF)GHI/	Inline comments allow notes to be placed within the RegEx. All comments will be ignored by the compiler. The example would match the string ABCGHI as the comment would not be compiled as part of the RegEx.
Internal Option Setting	/ABCD(?i)DEFG(?-i)/	The case sensitivity (i), multiline (m), dotall (s), and free-spacing (x) options can be toggled off and on within the rule. In the example ABCD will be case-sensitive whereas DEGF will be case insensitive.
Literal Strings	/ABCD/	A string of ASCII characters. The example will match the string ABCD.
Modifiers	/ABCD/six	Occur directly after the RegEx and affects the whole RegEx. This section provides more details on the modifiers supported by the RXP Compiler.
Non-capturing Parentheses	/(?:ABCD)*/	Specify grouping and precedence. In the example, the star symbol will apply to all characters contained within the parentheses. ?: will allow for parentheses to be explicitly defined as non-capturing.
Non-printing characters	/\t\n/	Control characters such as tab, newline etc. The example will match a tab followed by a newline.
Octal formats	/\377\377/	Use the \ delimiter to specify octal values. The example will match 2 bytes in a row each set to 255.
POSIX Character Classes	/AB[\n[:^digit:][:alpha:]]/	The POSIX notation for character classes is supported. The exact list of POSIX classes is locale dependent. The RXP Compiler supports the most widely used ones. The example will match first of all the text AB. This is then followed by a character class or what POSIX refers to as a bracket expression. This will match; newline, any non-digit or any alphabetic character.

Predefined Classes	/\d\s/	Groups of characters such as digits, whitespace etc. The example will match a digit character followed by a whitespace character.
Quoting	*\+ABC\Q?*+\E	Quoting will remove the special meaning from special characters. This can be achieved on single characters by preceding them with a backslash. If you wish to remove the special meaning from a sequence of characters they can be placed in between \Q and \E. The example will match the literal string *+ABC?*+.
Repetition	/A*B+C?/	Specifies multiple occurrences of any construct. The example will match zero to many occurrences of the letter A followed by one to many occurrences of the letter B followed by zero or one occurrences of the letter C.
Reset Subpattern Numbers	/(?|(AB)DE|(FG))\1/	This allows for the subpattern reference number to be reset for each captured alternation within the group. In the example, because the two alternatives are inside a (?| group they will both be numbered one. The example will therefore match both ABDEAB and FGFG.
User-defined Character Classes	/[ABCD0-9]/	Classes that represent a user-defined range. The example will match; A, B, C, D or any digit within the range zero to nine.
Word Boundary	/\bABCD\b/	Matches if the current position sits between a word and non-word character or start/end of job.

5.4. Alternation

Alternation is applied using the "|" (vertical bar) metacharacter, e.g. /(ABC|DE|FGHI)/ will match ABC, DE or FGHI. There is no restriction to the number of alternatives that appear, and an alternative may be empty e.g. /A(B|)/ is equivalent to /AB?/ and will match A or AB.

5.5. Anchored to Offset

This allows the match to be anchored to a specified offset from the beginning of the packet. This is represented by a dot metacharacter with a repetition value that must be placed directly after the symbol for the start anchor, e.g. /^.{4}AB/. If the anchored to offset construct is invalidly specified, it will follow the same rules as the repetition construct (see section Repetition). It is also supported to specify multiple offsets for clarity. These will all be merged into one anchor to offset structure. For example, /^.{4}.{8}AB/ is equivalent to /^.{12}AB/. The following table lists the anchor to offset metasequences supported by the RXP Compiler.

Table 15. Supported Anchored to Offset Operators

Metasequence	Description
^.{m, n}	Reference by number n (can be ambiguous with Octal notation (see Octal Formats))
^.{m}	Reference by number n
^.{m,}	Reference by number n
^..	Relative reference by number n

5.6. Anchoring

Anchoring can be achieved using the ^ and $ anchor metacharacters, and by the use of simple assertions. The multiline modifier (/m) is also supported, which affects the way the anchors are interpreted (see section Modifiers). The following table lists the anchoring formats supported by the RXP Compiler.

Table 16. Supported Anchors and Simple Assertions

Metasequence	Description
^	Anchor to start of subject and if in multiline mode, after newline also.
\A	Anchor to start of subject.
$	Anchor to end of subject and before newline at end of subject. If in multiline mode, anchor before any newline.
\Z	Always anchor to end of subject and before newline at end of subject.
\z	Always anchor to end of subject.

It is important to note that anchoring must be applied at a point in the expression where it can match the beginning or the end of the datastream (or match the beginning or end of a line if multi-line mode is enabled and the ^ or $ metacharacters are used). If this is not adhered to, the anchoring metacharacter will be invalid. An example of a RegEx that will not match anything would be /ABC^DEF/. It is also possible to apply anchoring to each alternation individually e.g. /(^ABC|DEF|^GHI$|JKL)/ or also to elements that occur after optional elements e.g. /(ABC)?(^DEF|GHI)/. In the previous example the DEF part of the alternation will only match if ABC does not occur before it. Valid matches would be ABCGHI, DEF and GHI but not ABCDEF.

5.7. Back References

Back references offer the ability to reuse a captured part of the RegEx match. The pattern that is back referenced is obtained by using capturing parentheses. The default method of back referencing is to use a backslash followed by a number greater than zero to invoke the back reference. This numeric value will increment for each set of capturing parentheses encountered in the RegEx. It is also possible to reference by name or relative reference. The table below lists the back-referencing formats supported by the RXP Compiler. When a back reference (named or numbered) is used in a RegEx it must be possible to pair it up with its referenced capture, otherwise an error will occur. All named back references must be less than 32 characters in size and can only contain alphanumeric characters and underscores and cannot begin with a number.

Table 17. Supported Back Referencing Styles

Back Reference	Description
\n	Reference by number n (can be ambiguous with Octal notation (see Octal Formats)).
\gn	Reference by number n.
\g{n}	Reference by number n.
\g{-n}	Relative reference by number n.
\k<name>	Reference by name (Perl notation).
\k'name'	Reference by name (Perl notation).
\g{name}	Reference by name (Perl notation).
\k{name}	Reference by name (.NET notation).
(?P=name)	Reference by name (Python notation).

The use of the \g sequence with a negative number signifies a relative reference. For example, /(ABC)(DEF)\g{-1}/ would match ABCDEFDEF and /(ABC)(DEF)\g{-2}/ matches ABCDEFABC.

An ambiguity exists with the "reference by number" type of back reference and the octal number format (see section Octal Formats). An example of this is \4: in theory this could be a back reference to the value captured in the fourth set of capturing parentheses or it could be the octal number four. To overcome this ambiguity the following rules apply in order of precedence:

1. Single digit escapes between \1 and \9 will always be interpreted as back references.
2. An escaped number beginning with zero is always an octal escape. E.g. \010 matches the "backspace" character.
3. If there is at least that number of previous capturing subpatterns, it will be taken as a back reference. E.g. \10 will be taken as a back reference if there are at least 10 sets of capturing parentheses before it. If there are not at least 10 sets of capturing parentheses, it will then be taken as the octal escape sequence for the "backspace" character.
4. Otherwise if the value is a qualifying octal number (\000 to \377) then the value will be taken as such.

These are discussed in the PCRE standard documentation and also referred to in the O'Reilly book "Mastering Regular Expressions". As back references are not supported in character classes, it is sufficient to simply infer that in this case any digit following a backslash will always represent an octal digit.

5.8. Capturing Parentheses

Besides grouping part of a RegEx together, round brackets also capture the part of the match that occurs within them which can then be used later as a back reference. The table below lists each of the capturing formats along with descriptions supported by the RXP Compiler. Groups will only capture data if the group has an associated back reference or conditional statement to use the captured data. If this is not the case, the group will be treated as non-capturing. All named captures must be less than 32 characters in size and can only contain alphanumeric characters and underscores and cannot begin with a number.

Table 18. Supported Capturing Styles

Capture	Description
(...)	Capturing group.
(?<name>...)	Named capturing group (Perl).
(?'name'...)	Named capturing group (Perl).
(?P<name>...)	Named capturing group (Python).

If a repetition (see section Repetition) value has been applied to a captured group, the captured value will be reset on all iterations of the loop and not appended to. An example of this is `/(A|B)*C\1/`, which will match `AACA` and `ABCB` but not `ABCAB`.

5.9. Conditional Statement

The conditional statement allows an if/then/else statement to be expressed and evaluated within a RegEx. The conditional statement is the form (?(condition)yes-pattern|no-pattern). The condition is always a back reference which is evaluated as to whether or not it has matched previously, yielding a boolean result. A simple example of the conditional statement is /1234(A)?(?(1)B|C)/, which will match 1234AB and 1234C but not 1234AC. The conditions can be expressed in various formats as shown in the table below. All named references must be less than 32 characters in size and can only contain alphanumeric characters and underscores and cannot begin with a number.

Table 19. Condition Styles

Condition Format	Description
(?(n)...)	Absolute reference condition.

(?(+n)...)	Positive relative reference condition. (not supported as requires forward reference).
(?(-n)...)	Negative relative reference condition.
(?(<name>)...)	Named reference condition (Perl).
(?('name')...)	Named reference condition (Perl).
(?(name)...)	Named reference condition (PCRE).

5.10. Dot Metacharacter

The dot metacharacter is supported. By default, it matches any ASCII character including newline to help maximize sustainable throughput. This can be overridden globally by using the RXP Compiler utility's –s option, then using the /s modifier on individual rules.

5.11. Inline Comments

This functionality permits comments to be interspersed with the RegEx, where they are used by the RegEx writer to help make the RegExes more understandable. Comments can be specified within the following construct (?#...). The compiler will ignore all comments when processing the input file e.g. /ABC(?#DEF)GHI/ will match ABCGHI.

5.12. Internal Option Setting

Internal option setting allows for features usually specified as mode modifiers (see section Modifiers) to be toggled on and off within the rule. These will usually modify the way the pattern matching operation should be performed. A list of supported internal options is given in the following table.

Table 20. Supported Internal Options

Modifier	Description
(?i)...(?-i)	This toggles on and off case insensitivity. This is functionally equivalent to the /i modifier.
(?m)...(?-m)	This toggles on and off multi-line mode. This is functionally equivalent to the /m modifier.
(?s)...(?-s)	This toggles on and off single-line mode. This is functionally equivalent to the /s modifier.
(?x)...(?-x)	This toggles on and off free-spacing mode. This is functionally equivalent to the /x modifier.

It is possible to specify multiple options in one statement, e.g. (?ix) to set case insensitive and free-spacing mode. It is also possible to combine the setting and unsetting of internal options such as (?ix-s) to set case insensitive and free-spacing mode and unset single-line mode. All internal options set within a set of parentheses will be turned off at the closing parentheses and the options that were set outside the parentheses will be reinstated. An example of this is /((?i)a)a/, which matches Aa or aa but not AA. It is also possible to specify a span for a sub-pattern where the options are set e.g. (?i-sx:sub-pattern) will match the sub-pattern inside the span with the options "i" and "x" turned on, and "s" turned off. The option settings will carry onto subsequent alternation branches even if the branch on which it occurs is not encountered during the matching process. The reason for this is that the option settings are all dealt with and applied at compile time and their span is applied to the two-dimensional "text" version of the RegEx without knowledge of the RegEx execution engine.

5.13. Hexadecimal Formats

The "\x" delimiter is used to signify hexadecimal values within rules. If the "\x" delimiter is encountered on its own, it will be interpreted as a hexadecimal escape with no following digits, giving a value of zero. Hexadecimal digits may be defined using upper and/or lower case letters. The following table describes the hexadecimal formats that are supported.

Table 21. Supported Hexadecimal Formats

Hexadecimal Format	Description
\xh.	h is a one-digit hexadecimal value representing a single character. Will be interpreted as \x0h.
\xhh	hh is a two-digit hexadecimal value representing a single character.
\x{hh}	hh is a two-digit hexadecimal value representing a single character.

5.14. Literal Strings

Strings of characters such as letters, digits, and special characters (including escaped with backslash) are supported. The characters will be bunched together where possible into long strings of an arbitrary length. The longest possible string will be extracted. The RXP Compiler will strive to form the largest possible strings to avail of the RXPs ability to process multiple characters per clock cycle.

5.15. Modifiers

Mode modifiers are operators appended to the end of a RegEx to modify the way the pattern matching operation should be performed. A list of supported RegEx modifiers along with descriptions is given in the following table.

Table 22. Supported Mode Modifiers

Modifier	Description
/i	If this modifier is set, letters in the pattern match both upper and lower-case letters in the subject string i.e. caseless or case insensitive. Caseless matching is only supported for characters with an ASCII value of less than 128. For caseless matching of characters with a value of greater than 128, Unicode must be supported.
/m	By default, PCRE treats the subject string as consisting of a single "line" of characters (even if it contains several newlines). The "start of line" metacharacter ^ matches only the start of the string, while the "end of line" metacharacter $ matches only at the end of the string, or before a terminating newline. When this modifier is set, the start of line and end of line constructs match immediately following or immediately before any newline in the subject string, respectively, as well as at the very start and end. If there are no "\n" characters in a subject string, or no occurrences of ^ or $ in a pattern, setting this modifier has no effect.
/s	Enables single-line mode. If this modifier is set, a dot metacharacter in the pattern matches all characters, including newlines. Without it, newlines are excluded. A negated class such as [^a] always matches a newline character, independent of the setting of this modifier.
/x	Enables free-spacing mode where all space characters (\x20) between RegEx tokens is ignored. It is important to note that only the space between tokens is ignored. In free-spacing mode the space character can be inserted to the RegEx by escaping it with a backslash or as part of a character class i.e. "\ " or "[ ]".

The RXP also has a set of custom modifiers that can affect the way each individual rule is compiled. A list of RXP custom RegEx modifiers along with descriptions is given in the following table.

Table 23. Supported RXP Custom Mode Modifiers

Modifier	Description
/c	If this is set then subpattern matching will be enabled for this rule.
/o	If this modifier is set, the rule will be split at its alternations.
/O	If this modifier is set, the rule will not be split at its alternations, even if the global switch is set.
/q	If this modifier is set, strict-quantifier mode will be used for this rule. To help with performance, by default the RXP Compiler treats non-fixed bounded quantifiers as unbounded e.g. .{0,2048} will be the same as .*. This has the caveat of false positives being possible. This modifier will ensure that the original construct is used for this meaning performance will be worse but no false positives will occur.
/Q	If this modifier is set, strict-quantifier mode will not be used for this rule, even if the global switch is set.
/p	If this modifier is set, the PTPB filter will be ignored for this rule.

5.16. Non-capturing Parentheses

Non-capturing parentheses are only used to group together parts of a RegEx and not used for capturing. By default, all parentheses will be interpreted by the RXP Compiler as non-capturing unless a back reference is paired with its captured data. Non-capturing parentheses can be explicitly expressed in the form (?:...).

5.17. Non-printing Characters

The "\" (backslash) character can be used to encode non-printing characters in a fashion that can be seen. Hexadecimal and octal notation can also be used to encode the characters however when used excessively they may obfuscate the RegEx. Inside a character class the "\b" metasequence represents the backspace character. The non-printing characters given in the following table are supported.

Table 24. Supported Non-Printing Characters

Non Printing Character	Hexadecimal Value	Description
\0	\x00	NULL.
\a	\x07	Alarm (BEL).
\cx (x = any alphabetic char)	Dependent on x	If x is a lower case letter it will be converted to uppercase. Then bit 6 of the character will be inverted. This feature is not available in XML mode.
\e	\x1B	Escape.
\f	\x0C	Formfeed.
\n	\x0A	Newline.
\r	\x0D	Carriage return.
\t	\x09	Horizontal tab.
\b	\x08	Backspace (only in character class).

5.18. Octal Formats

The following table lists all the octal formats along with a description that are supported by the RXP Compiler.

Table 25. Supported Octal Formats

Octal Format	Description
\d	d is a one digit octal value ranging from \0 to \7.
\dd	dd is a two digit octal value ranging from \00 to \77.
\ddd	ddd is a three digit octal value ranging from \000 to \377.

The RXP Compiler will attempt to extract the maximum number of octal digits immediately following an octal escape sequence to create a legal octal value (\000 to \377) indicating a single character.

There are issues arising with the \d, \dd and \ddd notation as it has an ambiguity with the "reference by number" back reference notation. See section Back References for more details on this and possible avoidance measures.

Note that if the \0 is immediately followed by non-legal octal digits, the \0 shall be interpreted as a NULL character (see section Non-printing Characters).

5.19. POSIX Character Classes

The POSIX classes given in the table below are supported. These can be inserted in character classes to denote a range of characters e.g. /ABC[0-9[:alpha:]]/ will match ABC followed by an alphanumeric character. The POSIX character classes that are available depend on the POSIX locale. The RXP Compiler supports all of the most widely used POSIX character classes. Note that POSIX classes include letters and digits defined in the locale, not just those in ASCII.

Table 26. Supported POSIX and Shorthand Equivalent Classes

Class	Equivalent	Description
[:alnum:]	[a-zA-Z0-9]	Alphanumeric characters.
[:alpha:]	[a-zA-Z]	Alphabetic characters.
[:ascii:]	[\x00-\x7F]	ASCII characters.
[:blank:]	[\x20\t]	Space or tab.
[:cntrl:]	[\x00-\x1F\x7F]	Control characters.
[:digit:]	[0-9]	Any decimal digit.
[:graph:]	[\x21-\x7E]	Any visible or printing character.
[:lower:]	[a-z]	Any lowercase alphabetic character.
[:print:]	[\x20-\x7E]	Visible characters including space also.
[:punct:]	[!"#$%&'()*+,\-./:;<=>?@[\\\]^_`{|}~]	Any punctuation character.

[:space:]	[\x20\t\r\n\v\f]	Any whitespace character.
[:upper:]	[A-Z]	Any uppercase alphabetic character.
[:word:]	[a-zA-Z0-9_]	Any word character.
[:xdigit:]	[0-9A-Fa-f]	Any hexadecimal digit.

5.20. Predefined Classes

The "\" (backslash) character is used to introduce predefined classes. The general type of predefined classes is used to specify a range of commonly used character classes. A list of these is given in the following table.

Table 27. Supported General Classes

Class	Equivalent	Description
\d	[0-9]	Any decimal digit.
\D	[^0-9]	Not a decimal digit.
\h	[\t\x20]	Any horizontal whitespace character.
\H	[^\t\x20]	Not a horizontal whitespace character.
\s	[\t\n\f\r\x20]	Any whitespace character.
\S	[^\t\n\f\r\x20]	Not a whitespace.
\v	[\n\x0B\f\r\x85]	Any vertical whitespace character.
\V	[^\n\x0B\f\r\x85]	Not a vertical whitespace character.
\w	[a-zA-Z0-9_]	Any word character.
\W	[^a-zA-Z0-9_]	Not a word character.

The RXP Compiler also has the option to use the XML schema RegExes predefined classes. There are four extra predefined classes supported that are not normally supported by any other flavor. The XML schema classes are shown in the following table.

Table 28. Supported XML Schema Classes

Class	Equivalent	Description
\i	[_:A-Za-z]	Any character that can be the first character of an XML tag name.
\I	[^_:A-Za-z]	Not a character that can be the first character of an XML tag name.
\c	[-._:A-Za-z0-9]	Any character that may occur after the first character of an XML tag name.

\C	[^-._:A-Za-z0-9]	Not a character that may occur after the first character of an XML tag name.

As can be seen from this table, the \c escape sequence is used to represent one of the XML predefined classes. This means when the XML schema mode is enabled the \c sequence no longer represents a control character escape sequence (see section Non-printing Characters) as it conflicts with the XML class notation.

5.21. Quoting

Quoting is used to remove the special meaning from characters and allows them to be treated as literals. Single metacharacters can be quoted by using the "\" (backslash) character e.g. /A*/ will literally match A*. If a literal backslash character is desired then the \\ sequence can be used i.e. a backslash quoting a backslash. Quoting using a backslash can only be used on single characters. Multiple characters can be quoted by surrounding them with \Q...\E e.g. /\QA*?+\E/ will literally match A*?+. It is important to note that the \E can be omitted at the end so /\QA*?+/ is equivalent to /\QA*?+\E/.

5.22. Repetition

Repetition can be applied to the following supported constructs:

- A literal character.
- The dot metacharacter.
- Any escape that matches a single character.
- A character class.
- A back reference.
- A parenthesized subpattern that is not an assertion.

Repetition is specified by quantifiers which specify a minimum and maximum number of permitted matches. The three most common quantifiers have been given single character abbreviations. All the repetition metacharacters (or metasequences, if they consist of more than one character) given in the following table are supported.

Table 29. Supported Repetition Operators

Repetition Metacharacters	Description
?	0 or 1 occurrences of previous construct.
+	1 or more occurrences of previous construct.
*	0 or more occurrences of previous construct.

{m, n}	Between m and n occurrences of previous construct.
{m}	Exactly m occurrences of previous construct.
{m,}	m or more occurrences of previous construct.

It is important to note that the RXP will treat all repetition as ungreedy. The quantifier {0} is permitted causing the expression to behave as if the previous construct and quantifier were not present. Infinite loops can be constructed by following a subpattern that can match no characters with a quantifier that has no upper limit e.g. /(a?)*/. If a repetition construct cannot be interpreted as valid, it will be interpreted as a literal string e.g. /A{,4}/ will match the string A{,4} or /A{1,4aa}/ will match the string A{1,4aa}. If the quantifiers are out of order i.e. the minimum repetition value is greater than the maximum repetition value, this will cause an error. Also if the repetition value exceeds the maximum repetition value of 32K, an error will be generated.

5.23. Reset Subpattern Numbers

This allows the subpattern reference number to be reset for each alternation e.g. /(?|(A)B|(C))\1/ will match ABA and also CC. This means that when the pattern matches, captured substring one can be used, regardless of which alternative matched. This can be used when it is desirable to capture part of one of a number of alternatives. The captures are numbered as normal inside a "reset subpattern numbers" group except the number is reset at the start of each alternation. E.g. in /(A)(?|(B)|(C(D))/ the captures noted in bold parentheses from left to right would be numbered 1,2,2,3.

5.24. Subpattern Matching

Subpattern matching can be switched on for a rule by using the "c" modifier. If subpattern matching is switched on then subpattern matches are reported alongside a full match, e.g. /A(B|C)DEFG/c will match ABCDEFG and also return a subpattern match of C.

5.25. User-defined Character Classes

The RXP Compiler is capable of supporting user-defined character classes. There are only certain metacharacters that are recognized within a user defined character class. The following table lists each of these along with their usage restrictions.

Table 30. Supported Character Class Metacharacters

Metacharacter	Description
[	The opening square bracket will begin the user-defined character class. It is recognized within the class as a metacharacter only when it is the beginning of a POSIX class (see section POSIX Character Classes).

]	The closing square bracket will terminate the user-defined character class. To use a closing square bracket as a literal member of a class it must either be escaped by a backslash '\', or occur directly after the opening square bracket. The closing bracket symbol cannot be used as the end character of a range.
^	The caret symbol can be used to negate the character class. This means that the subject must not match one of the class's members to be successful. To use the caret symbol as a literal member of the class it must either; be escaped by a backslash, or occur anywhere except directly after the opening square bracket.
\	The backslash symbol is used to remove the special meaning from characters and allows them to be treated as literals. The backslash can also be used within a character class to represent the RegEx literal items such as hexadecimal notation, octal notation and non-printing characters.
\Q...\E	Multiple characters can have their special meaning removed by surrounding them with \Q...\E (see section Quoting).
-	The minus symbol is used to specify ranges of characters. To use the minus symbol as a literal member of the class it must be escaped by a backslash or positioned in a place where it cannot be interpreted as indicating a range. The range of characters must be in ascending order e.g. [a-z] is valid whereas [z-a] is not.

There are standard methods of defining character classes using the metacharacters as discussed in the previous table. The combination of these metacharacters into the class notation is shown and discussed in the following table.

Table 31. Supported Character Class Notation

Class Notation	Description
[...]	Character class matching one of the characters contained within the square brackets.
[^...]	Negated character class matching any one character that is not contained within the square brackets.
[x-y]	Character class matching one of the characters in the range x to y.
[^x-y]	Negated character class matching any one character that is not in the range x to y.
[[:xxx:]]	Match any one character contained in the POSIX set xxx (see section POSIX Character Classes).
[[:^xxx:]]	Match any one character not contained in the POSIX set xxx (see section POSIX Character Classes).
[a-z-[aeiou]]	Character class subtraction is only available in XML schema mode and allows the matching of a character which is present in one list but not present in the subtracted list. The subtracted list must always be the last element in its containing character class e.g. [a-z1-4-[aeiou]] is valid but [a-z-[aeiou]1-4] is not. The subtraction will be applied to the entire class. The example shown in the class notation column will match any lowercase consonant i.e. by removing the vowels.

Nested character class subtraction is also supported. E.g. [0-9-[0-6-[0-3]]] first subtracts 0-3 from 0-6, yielding [0-9-[4-6]], or [0-37-9], which matches any character in the string 0123789.

The character class supports ranges of numerically specified characters. An example would be the use of hexadecimal notation to represent the character class [\x61-\x7A] or its equivalent in octal format [\141-\172] is also equivalent to the character class [A-Z]. It is also valid to use predefined classes within user-defined character classes. An example of this would be the use of the character class [\dA-Za-z] which will match any alphanumeric character and is equivalent to the \w predefined class.

The character class supports all of the non-printing characters. It also supports a special meaning for the \b escape sequence which usually means word boundary. This is used to represent the backspace character (\x08) in a character class.

5.26. Word Boundary

A word boundary metacharacter \b is used to determine that at that position in the RegEx if one character is a word character and the other is not, i.e. it sits across a word boundary, this is like the following RegEx:

```
(\w\W|\W\w)
```

Note that the word boundary also applies to start and end of a data stream i.e. anchors.

The negated version of the word boundary can be specified as \B.

5.27. Unsupported Constructs

The following table provides a brief description of each RegEx construct the RXP Compiler does not support.

Table 32. Regex Constructs Not Supported by the RXP Compiler

Construct	Example	Description
Anchor to start of match	/\G\d/	The anchor \G will match at the position where the previous match ended. This position will change each time the RegEx is applied to the subject string. If the example was applied to the string 1234A6 it could be applied successfully four times matching 1, 2, 3 and 4. It will fail on the fifth attempt because the only place where \G matches is on the 4 character which is followed by A. A is not in the \d class so the match will therefore be unsuccessful.
Atomic grouping	(?>\d+)ABC	An atomic group is a group that as soon as the RegEx engine exits from it, it automatically throws away all backtracking positions remembered by any tokens inside the group. If the RegEx in the example was \d+ABC, it would fail at 123456DEF and then the \d+ would then give up one

		match leaving it with 12345DEF. This would still fail and keep attempting the backtracking steps until all positions have been exhausted. If the subject 123456DEF was applied to the example it would fail immediately and not attempt to backtrack.
Backtracking control	/A(*ACCEPT)B/	These are verbs that act immediately when encountered. The example will match A and the verb (*ACCEPT) will cause the match to end successfully, skipping the remainder of the pattern.
Callouts	/ABCD(?C)E/	This allows external functions to be called within the RegEx. The example would match ABCD then a function would be called to perform some extra processing. Finally an E would result in a successful match.
Forward references	/(\2ABC|(DEF))+/	Forward references allow you to use a back reference to a group that appears later in the RegEx. The example will match DEFDEFABC. This is because DEF has to be captured before the \2 back reference will report a successful match.
Lazy quantifiers	/ABC*?/	As the RXP is ungreedy by default, the lazy quantifier will have no effect. In essence the RXP Compiler will convert all quantifiers to their lazy equivalent. The example will match AB as it will settle for the shortest match.
Lookaround assertions	/ABC(?=DEF)/	Lookahead and lookbehind do not consume any bytes. They will check to see if the specified pattern exists in front or behind the assertion. The example will match the string ABC if DEF occurs after it in the subject string.
Match point reset	/ABC\KDEF/	The escape sequence \K will cause any previously matched characters not to be included in the final matched sequence. In the example ABCDEF will be matched but only DEF will be reported.
Newline conventions	/(*CR)A.C/	These will override the default newline convention on the system and change it to the one specified. The RXP Compiler targets a Linux platform and uses "\n" as the default newline sequence. In the example a possible match would be A\nC. This is because the newline convention has been changed to carriage return.
Newline sequences	/ABC\R/	The escape sequence \R will match any newline sequence. It is equivalent to: (?>\r\n|\n|\x0b|\f|\r|\x85). The example will match ABC followed by any newline sequence.
Possessive quantifiers	/[^\n]*+D/	There is no backtracking available in possessive mode. If the RegEx in the example was greedy and had no possessive quantifier, it would fail at ABCE and then the [^ \n]* would then give up one match leaving it with ABC. This would still fail and keep attempting the backtracking steps until all positions have been exhausted. If the subject ABCE was applied to the example it would fail immediately.

Subroutine references	/ABC(?R)/	Subroutine references allow for subpatterns to be assessed as part of the RegEx. It is possible for these to be recursive. The example will match the string ABC infinite number of times as the (?R) means to recurse the whole pattern.
UTF-8, UTF-16 and UTF-32		The RXP Compiler does not support UTF-8, UTF-16 and UTF-32 in this release.
Unicode Properties	/ABC\X\p{Zl}/	The RXP Compiler does not support any of the UCP constructs: ▶ \p{xx} - a character with the xx property. ▶ \P{xx} - a character without the xx property. ▶ Caseless matching for characters > 128. ▶ \X extended Unicode sequence.

Chapter 6. Prefix Selection

The RXP uses a 1, 2, 3, or 4-byte "prefix" as the trigger for a match. It is extremely important WRT performance that a rule has a good prefix as this determines how many search threads are triggered. When a prefix is detected, one or more primary threads can be triggered. Each primary thread is dispatched to a Thread Engine (TE). The TE executes instructions associated with the search for a full match for one or more rules. A primary thread can trigger zero or more secondary threads during execution. These secondary threads are also managed by the TE. The RXP has a finite number of primary threads that can be executed at any one time. Due to this, any rules performance can be characterized for a set of data by the number of primary threads it generates for every byte.

In the rule above the prefix 1234 would be chosen with the up/down direction.

If rule_id=3 above was compiled with a denylist containing 1234 it would choose 3456 as a prefix with the direction down/up.

It is possible for a user to specify the prefixes they want a rule to use, although it is recommended to allow the RXP Compiler to automatically choose.

The RXP Compiler has many tools, heuristics and algorithms it uses when determining what prefix to use for a rule, some of these are discussed in the sections below. When writing rules for the RXP it is critical to use a good prefix, the sections below can be used to help do this.

6.1. Length

The length of a prefix is extremely important as it determines the probability this prefix will be hit in benign traffic. If a prefix is 1-byte then it will trigger one or more primary threads in:

```
1/256 bytes of uniformly random data
```

2-byte prefixes will trigger one or more primary threads in:

```
((1/256 * 1/256) = 1/65,536) bytes of uniformly random data
```

3-byte prefixes will trigger one or more primary threads in:

```
((1/256 * 1/256 * 1/256) = 1/16,777,216) bytes of uniformly random data
```

4-byte prefixes will trigger one or more primary threads in:

```
((1/256 * 1/256 * 1/256 * 1/256 *) = 1/4,294,967,296) bytes of uniformly random data
```

The benefits of a longer prefix can clearly be seen in the probability shown above. When the RXP Compiler is selecting a prefix, it will choose longer prefixes over shorter ones. By default, it will discard any one-byte prefixes, this filter can be overridden but if it is can be detrimental for a rulesets performance.

6.2. Context

The calculation in section Length does not take into consideration the context and assumes uniformly distributed random data. Data like Internet traffic will have more of a normal distribution, in this case the context of the target data is also important when selecting a good prefix. e.g. in Internet traffic strings like "http", "www", "get", "post" etc. will have a high incidence. From this contextual knowledge it can be postulated that these strings will be undesirable prefixes. The RXP Compiler uses a prefix selection control list to apply context to the prefix selection algorithms.

6.2.1. Prefix Selection Control List

The PSCL has three entry types:

1. Denylist:

   The RXP Compiler will not use any denylisted prefixes.

   If there are no other options, then the rule will not be compiled.
2. Graylist:

   The RXP Compiler will try to avoid using any graylisted prefixes.

   If there are no other options, the graylisted prefix will be used.
3. Allowlist:

   The RXP Compiler will try to use allowlisted prefixes first.

   If it can use any allowlisted prefixes it will select other prefixes.

A PSCL can be manually provided when compiling the ruleset, the RXP Compiler can also automatically generate one based on a sample of data or the rule content.

6.3. Multiple Prefixes per Rule

A rule can have many prefixes depending on how it is constructed, e.g if there are alternations and classes in a rule. The RXP Compiler will factor in the number of prefixes when it is choosing the position of the rule from where to extract the prefix. e.g. for the following rule:

```
1, /ABCD|1234/
```

Two prefixes will always be required, one for each alternation path i.e. ABCD and 1234. In the following rule:

```
1, /ABC[12]/
```

ABC could be used or else two prefixes including each class entry i.e. ABC1 and ABC2.

6.4. Jumpback

Jumpback can be used to permit alternative prefixes to be selected for a rule from within the first eight bytes. An example of where jumpback could be used is in the following rule:

```
1, /A[a-z][0-9]BCDEFG/
```

It is not preferable to use A as it is a one-byte prefix as discussed in section Length. It is also not preferable to extract all possible prefixes from the classes (see section Multiple Prefixes per Rule). The best choice for the RXP Compiler is therefore BCDE. In the rule above the RXP Compiler can choose BCDE as the prefix with a jumpback of three. This means that BCDE will trigger the search, jumpback three bytes, and then the RXP will process the RegEx.

6.5. Anchoring

The RXP has the capability to apply anchoring to specific byte pointers within the prefix engine. This serves to minimize the number of prefixes detected in a job. One important point to note when it comes to anchoring is that prefixes are shared between rules. This means that if multiple rules with different anchor values share a prefix, the anchoring in the prefix needs to be relaxed, for example in the following rules:

```
1, /^.{10}ABCD/
2, /^.{15}ABCD/
```

In rule_id=1, ABCD will be matched if it occurs at exactly byte pointer 10. In rule_id=2, ABCD will be matched if it occurs at exactly byte pointer 15. The two rules above will share the prefix ABCD, as there is only one anchor per prefix then this anchor will be relaxed to less than or equal to 15 in the prefix engine. As more rules get compiled the prefix can end up being completely unanchored in the prefix engine.

The strict anchor check will always be carried out after the prefix check when the RXP is processing the rest of the rule.

6.6. Unique Characters

For a good prefix it is important to ensure it has a diverse selection of characters. If a four-byte prefix contains all the same character, then it can turn into an extremely bad prefix. e.g. if the only available prefix is AAAA and there is a sequence of As in the data-stream the prefix AAAA will trigger on every byte.

6.7. Postfix

What comes after a prefix is also extremely important, e.g. for the following rule:

```
1, /ABCD.*12/
```

If ABCD is chosen as the prefix and is found, the postfix .* will then match anything until 12 is found. If 12 is never found, then the thread triggered by ABCD will stay alive right up until the end of the data stream. This can cause performance issues for larger jobs as it will take longer for the threads to die. This phenomenon is called "partial matches", more information on this can be found in section Optimization for .* Processing.

Chapter 7. Subset IDs

Rulesets can be split into one or more subsets of rules. Each subset of rules is given an identifier or subset_id. Jobs can then be scanned against up to four subsets from the rule set, rather than against all the rules in the rule set. For example, the following rule set is split into two subsets indicated by subset_ids = 1 and 2:

```
subset_id=1
1, ABCD
2, DEFG
subset_id=2
3, 1234
4, 5678
```

If the following job data is scanned against both subsets:

```
XXXXABCDXXXX1234
```

Matches will be detected for rule_ids 1 and 3. If the job data is scanned against only subset_id = 2, only the match rule_id = 3 will be detected. Each rule set can be split into up to 4,095 before subsets with a valid subset ID range of 1 to 4,095. Up to four subset IDs can then be presented per job so that the RXP scans job data against just those four subsets.

The prefix engine can do a partial subset ID check for each unique prefix to act as an additional filter. This partial subset ID check corresponds to the lower eight bits of the subset ID. The full 12-bit subset ID check is carried out in the thread engine. If a prefix is shared across more than one subset then the subset ID check can no longer be carried out in the prefix engine unless the target subset IDs share the lower eight bits.

For example, subset_id=1 is equivalent to subset_id=257, and subset_id=513 with regards to the prefix engine's partial subset ID filter.

For example, the following rule set is split into three subsets indicated by subset_ids = 1, 2, and 3:

```
subset_id=1
1, ABCD.*1234
subset_id=2
2, WXYZ
subset_id=3
3, ABCD.*5678
```

The prefix ABCD will be used for rules 1 and 3. As the prefix ABCD is shared across more than one subset, the subset ID check cannot be carried out in the prefix engine. The subset ID check will be carried out in the thread engine but the prefix engine subset filtering will be lost.

For example, the following rule set is split into three subsets indicated by subset_ids =1, 2, and 257:

```
subset_id=1
1, ABCD.*1234
```

```
subset_id=2
2, WXYZ
subset_id=257
3, ABCD.*5678
```

The prefix ABCD will be used for rules 1 and 3. This time the lower eight bits of the subset ID is also shared between the rules so the subset ID check can be carried out in the prefix engine.

Chapter 8. Differences Between RXP and PCRE

PCRE and the RXP have vastly different underlying architectures with one being software-based and the other hardware-based. Due to these differences, there are some situations when the outcome from the matching process also differs. These are not bugs however they are just down to differences in how the systems work. The following sections briefly describe a few examples of where this is the case.

8.1. Anchored to Offset

The RXP has a specialized construct for supporting anchors to offset. Due to the way this optimization is implemented the trigger point for the rule is after the anchor to offset construct. This can result in extra matches over what PCRE finds. For example, for the following rule and data:

```
1, /^.{0,5}ABCD/
ABCDABCD
```

PCRE will report the following match:

rule_id	start_ptr	Length
1	0	8

The RXP will report the following two matches:

rule_id	start_ptr	Length
1	0	4
1	0	8

In the case discussed above the RXP will trigger a search every time ABCD is encountered within the window specified by the anchor to offset construct. PCRE will only trigger one search at the beginning of the job.

Another example can be observed in is the following rule and data:

```
1, /^.?.?AB.*AB/
ABABXXXXXXAB
```

PCRE will report the following match:

rule_id	start_ptr	Length
1	0	4

The RXP will report the following two matches:

rule_id	start_ptr	Length
1	0	4
1	0	12

Another example of this can be observed in the following rule:

```
/^.{10,}A/
```

This rule is also implemented using our special anchored to offset construct, in PCRE this is just implemented as an anchored dot construct with a repetition.

This means that for the following job data:

```
AAAAAAAAAAAAAAAAAAAA
```

For the RXP, every byte from byte pointer 11 onwards triggers the prefix "A", whereas for PCRE there will only be one match spanning the entire job.

8.2. Relaxed Repetition Quantifiers

Constrained repetition can use up many resources when it occurs after a non-fixed repetition e.g. in the following rule:

```
1, /ABCD.*1234.{0,10}WXYZ/
```

The .{0,10} part of the above rule will require many resources and impact on performance to implement. As a performance vs. accuracy tradeoff the RXP Compiler will implement the above rule as the following:

```
1, /ABCD.*1234.*WXYZ/
```

This will perform much better but has a slightly increased chance of false positives or length mismatches.

There is a "strict-quantifiers" mode in the RXP Compiler that will disable this performance optimization in favour of accuracy, although this will also use many more resources and perform much slower.

The following table shows a list of rules with different permutations of repetition constructs. It shows what types of rules will have reduced performance if the strict quantifiers mode is used. If the strict quantifiers mode is not used then the constrained repetitions will be relaxed for those rules.

1	Max Bytes Consumed at a Time	Reduced Performance
/ABCD.*EFGHIJKL/	8	N
/ABCD.*EFGH.*IJKL/	8	N
/ABCD.*EFGH.{10}IJKL/	8	N

/ABCD.{10}EFGH.*IJKL/	8	N
/ABCD.{10}EFGH.{10}IJKL/	8	N
/ABCD.{0,10}EFGH.{10}IJKL/	8	N
/ABCD.{10}EFGH.{0,10}IJKL/	8	N
/ABCD.{0,10}EFGH.*IJKL/	8	N
/ABCD.* EFGH.{0,10}IJKL/	1	Y
/ABCD.{0,10} EFGH.{0,10}IJKL/	1	Y

8.3. Similar Alternation Paths

There are cases where the RXP will return identical matches e.g.:

```
/ABCD(EFGHIJKL|EFGHIJKL)/
```

The RXP will match both EFGHIJKL paths at the same time.

Due to this, the rule will always report two matches. If the rule has similar alternation paths, then multiple matches can be returned for an alternating rule at the same byte pointer.

8.4. Shortest Alternation Paths

There are cases when alternation branches are ambiguous when it comes to matching. The RXP will process all branches in parallel, so the shortest one will win. PCRE processes the branches sequentially from left to right, stopping as soon as one matches. e.g. for the following rule:

```
1, /ABCD(12|1|2)/
```

For the data ABCD12, the RXP will match ABCD1, whereas PCRE will match ABCD12.

8.5. Greediness

The RXP will always stop processing a primary thread as soon as a match is found. For example, in the following rule:

```
1, /ABCD.*/
```

The only possible match for this rule is ABCD, as soon as the RXP detects this then it will report a match. The shortest route through the .* is zero iterations so the RXP will take this path.

8.6. Start Pointer for Multi-line Mode

The RXP reports just before the newline as the start pointer whereas PCRE reports just after.

8.7. Repetitions at Beginning of Rule

If there is a repetition at the beginning of a rule concessions can be made in favour of being able to support the rule. For example, in the following rules:

```
1, /A*BCDE/
2, /A+BCDE/
```

If BCDE is chosen as a prefix, the unconstrained nature of the repetition means that jumpback cannot be used. In cases like these, the rule will be compiled using the minimum repetition value, this means that the shortest match will always be found. The rules above would be compiled like so:

```
1, /BCDE/
2, /ABCDE/
```

It is also important to note that when a rule starts with a + the '+' is removed it if is only one byte and kept if is related to more than 1 byte. It is also kept if it is one byte but anchored.

Chapter 9. Performance Considerations

There are certain considerations that need to be accounted for when writing rules for NVIDIA® RXP®, they are discussed in the following sections.

9.1. Complexity

The complexity of a rule refers to the number of alternation branches, repetitions, and complex structures like back references in the rule. If a rule is complex, then it will either not compile or have performance issues.

9.1.1. Rule Complexity

As a rule of thumb, the lower the complexity of a rule, the better it will perform. In terms of the RXP, rule complexity can impact resource usage and performance.

Resource usage is allocated on a per rule basis. Due to this it is prudent to split complex rules into multiple smaller rules where possible. The constructs that use most resources are backreferences and repetitions. If a rule has long sequences of these or nested repetition it may run out of resources at compile time or run time. NVIDIA RXP has a highly parallelized architecture, due to this, many small rules will perform better than fewer large rules.

Note that rules employing subpattern matching typically use a lot of resources. Due to this subpattern matching is limited to rules with a lower complexity.

9.1.2. Ruleset Complexity

As rules are compiled, they may not remain completely independent. The key component that is shared between rules and causes interactions are prefixes. As prefixes can be shared between rules the prefixes filtering effectiveness can decrease. A prefix can filter by subset ID but if multiple rules with different subset IDs share that prefix then the prefix can no longer check for the subset ID. A prefix can also check anchoring, but again as more rules share that prefix the anchor check needs to be relaxed in the prefix engine. See section Anchoring for more details on this. Note that even though the subset and anchoring checks are relaxed in the prefix engine, they are carried out in full in the thread engine.

9.2. Optimization for .* Processing

NVIDIA RXP has an optimization to allow .* constructs to be processed at up to eight bytes at a time. It is possible for this optimization to be active in a rule if it is followed by up to four different characters e.g. in the following rules:

```
1, /ABCD.*1234/
2, /ABCD.*(12|34)/
3, /ABCD.*[0-2]/
4, /ABCD.*[0-9]/
5, /ABCD.*(12|34|56|78|90)/
```

The optimization is possible for rule_id=1, rule_id=2, and rule_id=3. The .* construct in rule_id=1 is followed by only one character '1'. The .* construct in rule_id=2 is followed by two characters '1' and '3'. The .* construct in rule_id=3 is followed by three characters '0', '1', and '2.

The optimization is not possible for rule_id=4 and rule_id=5. The .* construct in rule_id=4 is followed by 10 characters, this exceeds the four character limit. The .* construct in rule_id=2 is followed by five characters which also exceeds the four character limit.

If the optimization cannot be used then the .* construct will fall back to consuming bytes one at a time.

9.3. Partial Matches

For example, in the following simple ruleset:

```
1, /AB/
2, /AB.*CD/
```

If there were a match for rule_id=1 embedded in a job, rule_id=2 would trigger a search thread. This thread would keep consuming bytes until either it matched with a CD or reached the end of the job. Rulesets constructed in such a way can keep NVIDIA RXPs processing engines busy. This phenomenon can be most apparent in larger jobs. If the .* avails of the optimization discussed in section Optimization for .* Processing, then the performance impact will not be as pronounced.