Generic CSV User Guide

Date: 28 January 2008

1 Overview

1.1 The Gateway Framework

The Generic CSV Gateway Kit uses the Gateway Framework as a container for the execution of its engine and post parser stages. The Gateway Framework and Generic CSV Gateway Kit are de-coupled into two separate installations. The Gateway Framework consists of a library of perl modules that provide functionality such as:

- a container for the execution of the Generic CSV Gateway Kit and Post Parser rules for of data transformation
- Intermediate (PIF) and output data (LIF,CSV,XML) storage and management
- logging utilities
- cleanup and crash recovery
- statistics gathering

The Generic CSV Gateway Kit simply plugs into the Gateway Framework and extends this base functionality to provide the final Gateway that parses a specific vendor's ASCII data.
More information on the standard Gateway configuration is contained in the Gateway Framework User Guide.
Only Generic CSV Gateway Kit configuration details will be described in this document.

1.2 Generic CSV Gateway Kit Overview

1.2.1 Data Types

The Generic CSV Gateway Kit can be configured to parse any ASCII data type through the use of both its native functions and optionally its pre-parser extension (Pre-Parser Configuration element).

The native functions provide all the basic rules for building a parser to work on most delimited data formats from the ground up, without needing to know perl language. To use the more advanced features such as the Pre-Parser element, (e.g. for more esoteric formats) you will need to have basic to intermediate knowledge of perl.

The formats allowed, ranges from basic CSV [or delimited] data (ideally suited to this Gateway for native, minimal configuration) to any other multi-line ASCII report format. Most data types/formats can be converted into a delimited representation by using the Pre-Parser element. The range of formats that have already been supported by the native functions is already large enough to consider the Generic CSV Gateway Kit the solution to most ASCII report data. Though there are some complex cases where a dedicated parser may be more appropriate.

The Pre-Parser element could be used to call external pre-parsers to parse other forms of data so long as the output sent back to the Engine is delimited data.

Included with the CSV Gateway Kit is the Pre Parser Simulator that enables you to develop the pre-parser solution outside of the Parser Framework.

1.2.2 Data Version Support

These are examples of systems currently using the Generic CSV Gateway Kit (up to v3.0.1.82) that exist and have been validated against the gateway ***without requiring*** *the use of the Pre-Parser Config element*.

Motorola SQL	GSR 7.0	SQL Informix multi-line
Nokia SQL	NetAct OSS 3.x	SQL Oracle multi-line
Alcatel SDH	1394	CSV + Topology
Lucent IN	8.5, 10.3	CSV data
Motorola IDEN	9.8, 10.5, 11.0	
Ericsson	J20	
Alcatel CN	-	
Nortel SDM		IP CSV Data
Metrica/NPR	Transfer Unload	NPR CSV Format

These are examples of systems that have been developed with the Generic CSV Gateway Kit 3.0.1.82 and above ***requiring*** *the use of the Pre-Parser Config element*.

Huawei GSM NSS	-	Non-CSV ASCII report

1.2.3 Data/File Formats

The Generic CSV Gateway Kit was developed on the basis of experience of PM report systems and allows any ASCII Format PM file (with delimited fields and line breaks) to be simply incorporated into a reusable and maintainable gateway.

It can be used in most cases without using the Pre Parser element where delimited records appear in the following formal format specification below. (In practical terms, the parser is highly optimised for this data format and can handle large data blocks up to the available RAM/SWAP limitation per process)

CSV File Format Formal Syntax:

```
{}      - group
[]      - optional
<>      - element
|       - sequence (followed by)
inf     - [theoretical] infinite number of
||      - or
&&      - and

<fileformat>    := { [ inf <commentline>|<headerline> ]
                       inf <datasection> }
<delimiter>     := <string constant>
<headerline>    := { <string> | [ inf <headerblock> ] }
<headerblock>   := { <delimiter><string> }
<dataline>      := { <string> | [ inf <datablock> ]
<datablock>     := { <delimiter><string> }
```

1.2.4 Architectural extensions

This parser adds 2 new files that **have to exist** for the correct operation of the parser though the functionality they provide can be turned off by not using them in the EngineConfig.pm. (providing back compatibility with versions of the CSV parser from v2 onwards)

- PreParserConfig.pm

  This file contains a set of perl subroutines (implemented as a hash of subroutines) that can be applied to input data to convert it into a delimited re-presentation of itself in RAM.

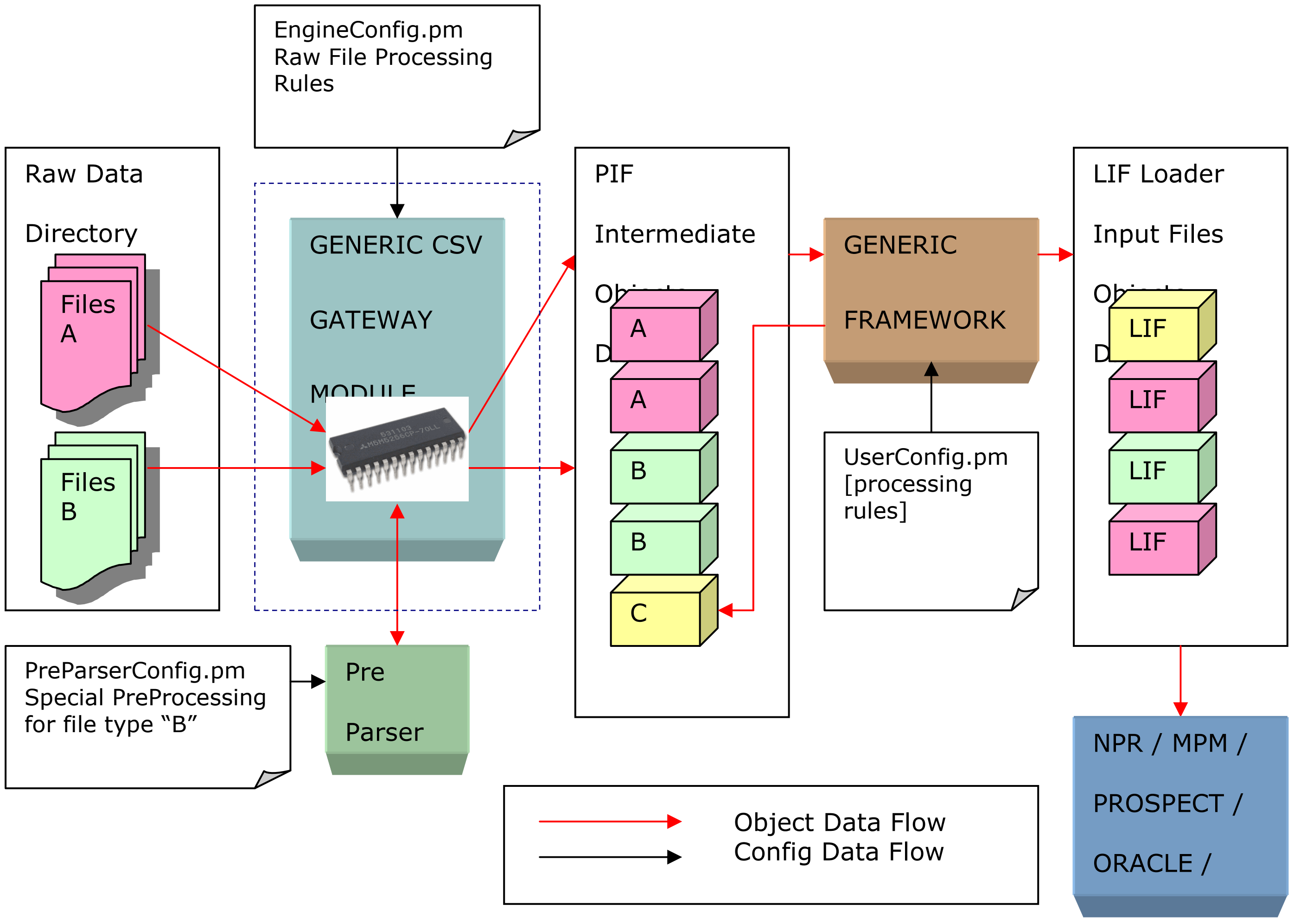
  Each subroutine defined in here should correspond to a particular data format that is to be converted. Details are provided in the supplied documentation and examples. This is provided to allow the user to define powerful functions to parse specific formats that would otherwise be impossible to use with the parser, or require new parsers to be written.

- PreParserSimulator

  This is provided to simulate a selected pre parser action on a raw data file and reproduce the RAM representation on your standard terminal. Use this when developing new pre-parsers. Your format will be output as delimited data and the field positions chosen in the EngineConfig.pm should reflect this output when using the PRE_PARSER directive in the EngineConfig.pm

1.3 Processing Diagram

File type A is a normal CSV or delimited file, File type B is a complex file type that requires further processing to turn it into CSV data. Two "rules" are defined for File type "A" & "B" in the EngineConfig.pm. The File type "B" is a complex non-CSV format and the EngineConfig.pm employs the PRE_PARSE directive to modify the state of the loaded file type "B" in memory and then process them according to the EngineConfig.pm "B" rule. Then the PIF objects are created & processed by the framework's Post Parser which uses UserConfig.pm to convert the generic format PIF's into LIF, CSV or XML output.



2 Engine Rules and Configuration

Each file format you want to parse with the Generic CSV Gateway Kit must have its own Engine Rule entry containing the set of Engine Configuration options described below. Some Engine Configuration options are mandatory and some will be optional depending on the file format that the Engine Rule is based upon.

2.1 RULE_TYPE

Describes the Engine Rule (gateway) to use to parse a specific file format. If set to "GENERIC_CSV_GATEWAY", the Engine Configuration tells the gateway framework to use the generic csv gateway engine. After this is set correctly the other Engine Configuration Options listed below then apply to the generic csv gateway only.

Any Single Rule Type entry calls a specific gateway engine to convert a particular input file format to the parser intermediate format (PIF). Each Configuration will be different for different input formats.

2.1.1 Rule Configuration

The following details the vendor specific rule entries for the Generic CSV engine rule.

- RULE_TYPE: Mandatory entry.

```
RULE_TYPE   =>    'GENERIC_CSV_GATEWAY',
```

- RULE_DESC: Simple Free form text to describe what format this Rule Entry will cater for. Mandatory entry.

```
RULE_DESC   => '(free text) file format of this data',
```

- INPUT_FILE_DESCRIPTION: Specification for Input files in the specified property network raw data directory "IN_DIR" (e.g. '^.*.csv$') is a regular expression which denotes all files which end in the ".csv" extension which are present in the input directory. More than one set can be specified in the []'s

```
INPUT_FILE_DESCRIPTION  => [ '^.*.csv$' ],
```

- INPUT_DIR_DEPTH: Specification for how deep the gateway framework will search for files in directory trees in the specified property "IN_DIR". 0 is the default. Mandatory entry.

```
INPUT_DIR_DEPTH         => '0',
```

- NUMBER_OF_FILES_TO_PROCESS: Specification for how many files from "IN_DIR" the gateway framework will attempt to run through the gateway in any single run. E.g. 20 = 20 files at a time. This parameter is a tuning parameter. Mandatory entry.

```
NUMBER_OF_FILES_TO_PROCESS    => '20',
```

- ORDER_OF_FILES: Specification for what order the gateway framework will attempt to parse the data through the gateway. No ordering is fastest, oldest first is best for PM files due to NC re-parenting. Mandatory entry.

  Valid options: YOUNGEST_FIRST, OLDEST_FIRST, DIRECTORY_ORDER (comment out this configuration option to speed up the first stage of the parser at expense of data presentation contiguity to the loader). This parameter is a tuning parameter.

```
ORDER_OF_FILES    => 'YOUNGEST_FIRST',
```

- HEADER_FIELDS: HEADER_FIELDS numbers the fields in the raw input data that the user wants to appear in the header of PIF/LIF. It is using an array of numbers to represent the header field positions in the input data counting from left to right from zero.

  Header fields are usually those that are common to every block under it
  All fields start from 0
  e.g. if you want nodeid in header and nodeid is field 0, then
  specify [0], If you want nodeid, day and day is field 1, then
  specify [0, 1], or [0 .. 1],
  (required in non-topology mode) (optional in topology mode)

```
HEADER_FIELDS          => [ 0 .. 1 ],
```

- DATA_FIELDS: DATA_FIELDS numbers the fields that the user wants to appear in the blocks of the PIF/LIF by using an array of numbers to represent the header field positions in the input data format. Mandatory entry.

  Data field numbers the fields that you want to appear in blocks of lif
  All fields start from 0
  e.g. if you want time in lif block and time is field 4, then
  specify [4], if you want all fields from 10-15 and 1-4 then
  specify [1 .. 4, 10 .. 15],
  (required)

```
DATA_FIELDS       => [ 1 .. 4, 10 .. 15 ]
```

- FIELDS_TO_KEY_PIF_FILENAME: FIELDS_TO_KEY_PIF_FILENAME numbers the fields that the user wants to use as the PIF filename key. PIFs appear in the intermediate directory "INT_DIR" property directory as individually named files. This controls what the intermediate filename will be named as.

  This allows you to batch all data into one file, or split it up into several smaller ones depending on the fields you choose. The more fields you add, the more PIF files you produce per input file (up to the max open files per process parameter). In Metrica/NPR it's better to have NODEID, DAY and match the retrieval key index of the loadmap to the pif filename, and put NODEID, DAY into the HEADER_FIELDS option also. (This is normally required for best operation, though it can be commented out to use the original filename as the PIF name) Assuming NODEID is in FIELD 0 and DAY is in FIELD 1 in the data file PM.csv:
  ```
  FIELDS_TO_KEY_PIF_FILENAME    => [ 0 , 1 ],
  ```

  We created a file called NODEID-#-DAY-#-PM.csv-#-I.pif in "INT_DIR"
  ```
  FIELDS_TO_KEY_PIF_FILENAME    => []
  ```

  We created a file called PM.csv-#-I.pif in "INT_DIR"

- DATE_TIMESTAMP_FIELD: This option has to be configured along with its complimentary options:
  - DATE_TIMESTAMP_FIELD: Array of fields to process (array of field numbers)
  - DATE_TIMESTAMP_TYPE: Array of corresponding field type identifier strings
  - DATE_TIMESTAMP_OPTS: Array of corresponding field type conversion strings

  This allows the user to use the built-in functions to modify a date field from one date or time format into standard Metrica/NPR / Performance Manager date & time formats. They can be mixed and matched as required, though all 3 elements are usually necessary.
  POSIX - The option POSIX is limited to converting posix (unix) time stamps to standard date/time string representations. The below example takes fields 1 and 2 and creates new header records in the LIF. The contents of fields 1 and 2 in the input data are a posix time. An example resulting header counter output is POSIX0_0 14May01 and POSIX0_1 12:00 which processed column 0 below, and POSIX1_0 14May01 and POSIX1_1 11:00 for column 1 below. The option DATE_TIMESTAMP_OPTS needs to be set to GMTIME if you want no local time zone offset from the POSIX timestamp, or it will apply the time zone to the POSIX time using the local time zone by default. (Desirable in some cases)
  ```
  DATE_TIMESTAMP_FIELD => [       1,       2 ],
  DATE_TIMESTAMP_TYPE  => [ 'POSIX', 'POSIX' ],
  DATE_TIMESTAMP_OPTS  => [  'NONE','GMTIME' ],
  ```

  TIMEROUND – The option TIMEROUND rounds integer durations up/down to nearest n seconds where n=OPTS) The example below would round 901 to 900, 850 to 900, and 1805 to 1800. This is useful when data is reported by crontab, or there is a script that runs that collects data and doesn't quite report its start time correctly.
  ```
  DATE_TIMESTAMP_FIELD   => [ 1 ],
  DATE_TIMESTAMP_TYPE    => [ 'TIMEROUND' ],
  DATE_TIMESTAMP_OPTS    => [ 900 ],
  ```

STRINGDMY, STRINGMDY, STRINGYMD – These options are for dealing with dates in the form "10-05-2004" or "10_05_04" or other delimited string date formats. Pick the correct order for your date string. UK dates would be DMY for example and YMD for US. Output is transformed to preferred format (e.g. 10May2004) The delimiter is a mandatory option and can be as complex as you need to break up the string into its components.

```
DATE_TIMESTAMP_FIELD => [ 1 , 2 , 3],
DATE_TIMESTAMP_TYPE  => [ 'STRINGDMY', 'STRINGMDY', 'STRINGYMD'
],
DATE_TIMESTAMP_OPTS  => [ "-" , "_" , "." ],
```

- INVALID_VALUE_MATCH: Removes all the items from the raw input data file which match the characters listed before the gateway engine processes it.

```
INVALID_VALUE_MATCH     => [ "'", '"' ],
```

- FIELD_SEPARATOR: This is the Key to the field splitting process and controls the fields that are used internally. The field separator delimits your data. Mandatory entry.

```
FIELD_SEPARATOR   => ‘,’,
```

If you have "A;B;C" then use ";" .. if you have "1,3,65,2" then use "," .. if you have a weird format such as "ABC##456##928" then use "##".. most things are possible.. including tabs "\t". See the documentation provided with the parser for more details.

- WHITESPACE_MODIFIER: This option allows the user to change any “ “ (space) characters to underscores or other more useful character.

```
WHITESPACE_MODIFIER   => '_',
```

- HEADERLINES_TO_SKIP: This option allows the user to throw out any number of header lines from the top of the file, to discard unwanted information, such as report dates and times etc.

```
HEADERLINES_TO_SKIP   => 2,
```

This would skip two blank or unwanted lines before the “real” data started in the file.

- FAST_VALIDITY & SECURE_VALIDITY: This checks the file before processing and only processes the 1st two lines or the whole structure (if SECURE_VALIDITY is used). Optional entries.

```
FAST_VALIDITY     => '.*;.*',

SECURE_VALIDITY   => '.*;.*',
```

Fast checks are performed before the optional header line is placed is placed onto the top of the file by the HEADER_OPTION. Secure checks are performed on all lines. multiple checks can be added by using the | operator within the regular expression.

- HEADER_OPTION: This option allows the user to add in the description (counter names) of the data fields if they are not present. (if they are present, please do not use it!). The array contains the names of the counters that you want to represent your field positions. Optional entry.

```
HEADER_OPTION => ['SCHEDTIME', 'ACTUALTIME', 'DURATION',
'MOUNTED_ON', 'KBYTES', 'USED', 'SCP'],
```

- LINE_DELIMITER: Optional field. This allows you to split the file into records based on a different delimiter than new line (\n) e.g. setting it to ';' would read multi-lines ending with a semi-colon. Setting invalid value match to "\n" allows complex multiline datasets to be read in, in one pass.

```
LINE_DELIMITER        => '\n',
```

- NORM_DELIMITER: Optional Field. (Norm means: "normal", i.e. what it should be set to normally) This is needed if using LINE_DELIMITER to restore the original linefeed character "\n". You can also parse strange records by using combinations of the LINE_DELIMITER and NORM_DELIMITER.

```
NORM_DELIMITER        => '\n',
```

- GLOBAL_SUBSTITUTION: Global substitution applies to every line in the data file before parsing occurs. The example below will replace occurrences of DURATION with TIMESPAN and NECELLID with CELLID in the file. You can keep adding substitutions without affecting too much the performance of the engine itself.

```
GLOBAL_SUBSTITUTION => { DURATION => 'TIMESPAN',
                         NECELLID => 'CELLID', },
```

- UPPERCASE_COUNTERS: Uppercase counters should be set to True for all previous parser releases to V3.0.073 to maintain compatibility. On Performance manager systems you may not want the uppercase function (as the loadmap is case sensitive). Set to '0' to disable, and 'True' to enable.

```
UPPERCASE_COUNTERS    => 'True',
```

- PIF_FILENAME_EXTRACT: PIF FILENAME EXTRACT operates on the raw filename and extracts a new PIF filename based on the regular expression you choose. The fields you choose in brackets "()" which match will be joined with "." characters. This rule is mainly for cutting the size of the file down or making new filenames from existing name and user fields.

```
PIF_FILENAME_EXTRACT   => '(\w+)\.',
```

E.g: File = bss_statistics.omcsys2.02Jul2004.21:50:49.27873.Z
Rule = 'PIF_FILENAME_EXTRACT' => '(\w+)\.(\w+)\..*'
PIF Name = <FIELDS_TO_KEY_PIF_FILENAME>-#-bss_statistics.omcsys2-#-I.pif
(Note that the date and time has been removed from the PIF)

- JUNK_MATCH: This rule will remove entire comment lines containing the regular expression before parsing starts. Without affecting the input file contents. The example below removes any comment line starting with "#"

```
JUNK_MATCH            => '^#.*$',
```

- TOPOLOGY_MODE: Will turn on the new TOPOLOGY mode which emulates the behaviour of the CSV_GENERIC_TOPOLOGY_GATEWAY.pm parser which used to ship with versions prior to 3.0.175. Now you **must** use GENERIC_CSV_GATEWAY for all RULE_TYPE entries in your parser. There are no longer separate parser engines for the CSV data and Topology data.

If converting Engine Configurations from releases prior to 3.0.175 you must specifically set the data fields to output by adding 'DATA_FIELDS' to the EngineConfig entry for your parser. There is no need to have 'HEADER_FIELDS' set when TOPOLOGY_MODE is used as HEADER_FIELDS is now an optional element when in topology mode.

```
TOPOLOGY_MODE         => 'True',
```

- PRE_PARSE: This rule executes the selected pre parser function (from the PreParserConfig.pm) on all input files that are specified by this Engine Rule's INPUT_FILE_DESCRIPTION. This will process your input file before it is processed by the main CSV engine. To see the output of the pre parser you will need to use the PreParserSimulator command.

  ```
  PRE_PARSE     => 'MYVENDOR_VER',
  ```

  More technical documentation is located in the accompanying documentation. 3.0.1.80 & 3.0.1.81 versions of the parser should be migrated to use the new PreParserConfig.pm and any subroutines should be sent to the gateways group for possible product inclusion.

- QUICK_DATA_SPLIT: This rule will speed up parsing provided the data is simple CSV:
  - the field separator consists of a simple character e.g. ',’
  - quoted strings are escaped with pairs of '"' e.g. "here, this is an example"

  ```
  QUICK_DATA_SPLIT            => 'True',
  ```

- OVERWRITE_PIF: Enable this rule if existing PIFs are to be overwritten (this overrides the default framework behaviour). This rule is helpful when parsing configuration files which are retained whose PIF files are retained. The PIFs are then replaced when fresh configuration files are available.

  ```
  OVERWRITE_PIF => 'True',
  ```

3 Post Parser Rules and Configuration

Only standard Post Parser Rules Are Supplied. There are no vendor specific rules as the parser is not vendor specific, only the configurations of the Gateway engine are vendor specific.

4 Installation Specific Information

The parser (for speed & functionality purposes) sometimes keeps many files open at once during the process of parsing each input file. It will not report when it goes over the user limit of maximum open files per process in this release. Normally this will not be a problem unless as a user you have set FIELDS_TO_KEY_PIF filename to generate many PIF files per input file. It is better to keep the PIF files to minimum as it reduces the parsing overhead at the POST PARSER stage.
There are ways to increase the maximum user limit (which is normally 256):
On HP-UX systems within the bourne shell the maximum files per process limit can be increased by using ulimit –n X, where X is the no of files per process. You can set this in the gateway_start.sh script.

On Solaris, the OS contains a bug that limits its interoperability to 256 files and any changes to the ulimit will be ineffective.