ffe

flat file extractor
Version 0.3.7, 22 January 2017

by Timo Savinen

This file documents version 0.3.7 of `ffe`, a flat file extractor.

1 Preliminary information

The `ffe` is a program to extract fields from text and binary flat files and to print them in different formats. The input file structure and printing definitions are specified in a configuration file, which is always required. Default configuration file is `~/.fferc` (`ffe.rc` in windows).

`ffe` is a command line tool developed for GNU/Linux and UNIX systems. `ffe` can read from standard input and write to standard output, so it can be used as a part of a pipeline.

There is also binary distribution for windows.

2 Samples using ffe

One example of using ffe for printing personnel information in XML format from fixed length flat file:

```
$ cat personnel
john     Ripper       23
Scott    Tiger        45
Mary     Moore        41
$
```

A file personnel contains three fixed length fields: 'FirstName', 'LastName' and 'Age', their respective lengths are 9,13 and 2.

In order to print data above in XML, following configuration file must be available:

```
$cat personnel.fferc
structure personel {
    type fixed
    output xml
    record person {
        field FirstName 9
        field LastName  13
        field Age 2
    }
}

output xml {
    file_header "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n"
    data "<%n>%t</%n>\n"
    record_header "<%r>\n"
    record_trailer "</%r>\n"
    indent " "
}
$
```

Using ffe:

```
$ffe -c personnel.fferc personnel
<?xml version="1.0" encoding="ISO-8859-1"?>
 <person>
  <FirstName>john</FirstName>
  <LastName>Ripper</LastName>
  <Age>23</Age>
 </person>
 <person>
  <FirstName>Scott</FirstName>
  <LastName>Tiger</LastName>
  <Age>45</Age>
 </person>
 <person>
```

```
  <FirstName>Mary</FirstName>
  <LastName>Moore</LastName>
  <Age>41</Age>
 </person>
$
```

3 How to run ffe

ffe is a command line tool. Normally ffe can be invoked as:

```
ffe -o OUTPUTFILE INPUTFILE...
```

ffe uses the definitions from the configuration file and tries to guess the input file structure. If the structure cannot be guessed the option -s must be used.

3.1 Program invocation

The format for running the ffe program is:

```
ffe option ...
```

ffe supports the following options:

`-c` *file*
`--configuration=`*file*
: Configuration is read from *file*, instead of `~/.fferc` (`ffe.rc` in windows).

`-s` *structure*
`--structure=`*structure*
: Use structure *structure* for input file, suppresses guessing.

`-p` *output*
`--print=`*output*
: Use output format *output* for printing. If not given, then the record or structure related output format is used. Printing can be suppressed using format *no*. Original data is printed using format *raw*.

`-o` *file*
`--output=`*file*
: Write output to *file* instead of standard output.

`-f` *list*
`--field-list=`*list*
: Print only fields and constants listed in the comma separated list *list*. Order of names in *list* specifies also the printing order.

`-e` *expression*
`--expression=`*expression*
: Print only those records for which the *expression* evaluates to true.

`-a`
`--and`
: Expressions are combined with logical and, default is logical or. Note that if the same field and operator appear several time in expressions they are always compared with logical or.

`-X`
`--casecmp`
: Expressions are evaluated using case insensitive comparison

`-v`
`--invert-match`
: Print only those records which don't match the expression.

`-l`
`--loose` Normally `ffe` stops when it encounters an input line or binary block which doesn't match any of the records in selected structure. Defining this option causes `ffe` continue despite the error. Note that invalid lines are reported only for text input. In case of binary input next valid block is silently searched.

`-r`
`--replace=`*field*`=`*value*
Replace *field*s contents with *value* in output. *value* can contain same directives as output option `data`.

`-d`
`--debug` All invalid input lines are written to `ffe_error_<pid>.log`, where `<pid>` is the process ID.

`-I`
`--info` Show structure information in the configuration file and exit successfully. For every structure following information in shown:
Structures: Name, type and maximum record length.
Records: Name and length
Fields: Name, position and length. First position is number one.

`-?`
`--help` Print an informative help message describing the options and then exit successfully.

`-V`
`--version`
Print the version number of `ffe` and then exit successfully.

All remaining options are names of input files, if no input files are specified or `-` is given, then the standard input is read.

Expressions (option `-e`, `--expression`)

Expression can be used to select specific records comparing field values. Expression has syntax *field*`x`*value*, where `x` is the comparison operator. Expression is used to compare field's contents to *value* and if comparison is successful the record is printed. Several expressions can be given and at least one must evaluate to true in order to print a record. If option `-a` is given all expressions must evaluate to true.

If *value* starts with string `file:` then the rest of *value* is considered as a file name. Every line in file is used as *value* in comparison. Comparison evaluates true if one or more values matches, so this makes possible use several different values in comparison. **Note**: The file size is limited by available memory because the file contents is loaded to memory.

When comparing binary fields the *value* must have the representation which can be shown using the `%d` output directive. Note that the printing option *hex-caps* takes effect in comparison.

Expression notation:

field=*value*
Field *field* is equal to *value*.

field^*value*
: Field *field* starts with *value*.

field~*value*
: Field *field* contains *value*.

field!*value*
: Field *field* is not equal to *value*.

field?*value*
: Field *field* matches the regular expression *value*. `ffe` supports POSIX extended regular expressions.

3.2 Configuration

`ffe` uses configuration file in order to read the input file and print the output.

Configuration file for `ffe` is a text file. The file may contain empty lines. Commands are case sensitive. Comments begin with the `#`-character and end at the end of the line. The `string` definitions can be enclosed in double quotation `"` characters. `char` is a single character. `string` and `char` can contain following escape codes: `\a`, `\b`, `\t`, `\n`, `\v`, `\f`, `\r`, `\"` and `\#`. A backslash can be escaped as `\\`.

Configuration has two main parts: the structure, which specifies the input file structure and the output, which specifies how the input data is formatted for output.

Common syntax

Common syntax for configuration file is:

```
#comment
‘command‘
const name value
filter name value
...
structure name {
    option value ...
    ...
    record name {
        option value ...
        ...
    }
    record name {
        option value ...
        ...
    }
    ...
}
structure name {
    ...
}
...
output name {
    option value ...
```

```
    ...
}
output name {
    ...
}
...
lookup name {
    option value ...
    ...
}
lookup name {
    ...
}

...
```

Structure

Keyword `structure` is used to specify the input file content. An input file can contain several types of records (lines or binary blocks). E.g. file can have a header, data and trailer record types. Records must be distinguishable from each other, this can be achieved defining different 'keys' (`id` in record definition) or having different line lengths (for fixed length) or different count of fields (for separated structure) for different records.

If binary structure has several records, then all records must have at least one key (`id`), because binary blocks can be distinguished only by using keys.

The structure notation:

```
structure name {
    option value ...
    ...
}
```

A structure can contain following options:

`type fixed|binary|separated [char] [*]`
: The fields in the input are fixed length fields (text or binary) or text fields separated by *char*. If * is given, multiple sequential separators are considered as one. Default separator is comma.

`quoted [char]`
: Fields may be quoted with char, default quotation mark is the double quotation mark '`"`'. A quotation mark is assumed to be escaped as *char* or doubling the mark as *charchar* in input. Non escaped quotation marks are not preserved in output.

`header first|all|no`
: Controls the occurrence of the header line. Default is no. If set as *first* or *all*, the first line of the first input file is considered as header line containing the names of the fields. *first* means that only the first file has a header, *all* means

means that all files have a header, although the names are still taken from the header of the first file. Header line is handled according the record definition, meaning that the name positions, separators etc. are the same as for the fields. Binary files cannot have a header.

`output` *`name`*`|no|raw`
: All records belonging to this structure are printed according output format name. Default is to use output named as '`default`'. '`no`' prints nothing and '`raw`' prints only the original data.

`record` *`name`* `{`*`options ...`*`}`
: Specifies one record for a structure. A structure can contain several record types.

Record

A record specifies one type of input line or binary block in a file. Different records can be distinguished using the `id` option or different line lengths or field counts. In multi-record binary structure every record must have at least one `id` because binary records do not have a special end of record marker as text lines have.

The record notation:

```
record name {
    option value ...
    ...
}
```

A record can contain following options:

`id` *`position string`*
`rid` *`position regexp`*
: Identifies a record in the input file. Records are identified by the *string* or by the regular expression *regexp* in input record position *position*. For fixed length and binary input the position is the byte position of input record and for separated input the *position* is the *position*'th field of the input record. Positions starts always from one.

  A record definition can contain several id's, then all id's must match the input line (`id`'s are *and-ed*).

  Non printable characters can be escaped as '`\xnn`', where '`nn`' is characters hexadecimal value.

`field` *`name`*`|FILLER|* [`*`length`*`]|* [`*`lookup`*`]|* [`*`output`*`]|* [`*`filter`*`]`
: Defines a field in a text input structure. *length* is mandatory for fixed length input structure.

  The last field of a fixed length input structure can have a * in place of *length*. That means that the last field has no exact length specified and it gets the remainder of the input line after all other fields. This allows a fixed record to have arbitrary long last field.

  Length is also used for printing the fields in fixed length format (directive `%D` in output definitions).

If * is given instead of the name, then the *name* will be the ordinal number of the field, or if the `header` option has value *first* or *all*, then the name of the field will be taken from the header line (first line of the input).

If *lookup* is given then the fields contents is used to make a lookup in lookup table *lookup*. If *length* is not needed (separated format) but lookup is needed, use asterisk (*) in place of length definition.

If *output* is given the field will be printed using output definition *output*. If *length* and/or *lookup* are not needed use asterisk in place of them. Use asterisk (*) if not needed.

If *filter* is given the raw contents of the field is filtered through a program defined by *filter* and the output of the program is printed as field contents.

If field is named as `FILLER`, the field will not appear in output.

The order of fields in configuration file is essential, it specifies the field order in a record.

`field` *name*`|FILLER|*` *length*`|`*type* [*lookup*]`|*` [*output*]`|*` [*filter*]

Defines a field in a binary structure. All other features are same as for text structure fields except the *type* parameter.

type specifies the field length and type and can have the following values:

`char`	Printable character.
`short`	Short integer having current system length and byte order.
`int`	Integer having current system length and byte order.
`long`	Long integer having current system length and byte order.
`llong`	Long long integer having current system length and byte order.
`ushort`	Unsigned short integer having current system length and byte order.
`uint`	Unsigned integer having current system length and byte order.
`ulong`	Unsigned long integer having current system length and byte order.
`ullong`	Unsigned long long integer having current system length and byte order.
`int8`	8 bit integer.
`int16_be`	Big endian 16 bit integer.
`int32_be`	Big endian 32 bit integer.
`int64_be`	Big endian 64 bit integer.
`int16_le`	Little endian 16 bit integer.
`int32_le`	Little endian 32 bit integer.
`int64_le`	Little endian 64 bit integer.
`uint8`	Unsigned 8 bit integer.
`uint16_be`	Unsigned big endian 16 bit integer.

`uint32_be`
: Unsigned big endian 32 bit integer.

`uint64_be`
: Unsigned big endian 64 bit integer.

`uint16_le`
: Unsigned little endian 16 bit integer.

`uint32_le`
: Unsigned little endian 32 bit integer.

`uint64_le`
: Unsigned little endian 64 bit integer.

`float` Float having current system length and byte order.

`float_be` Float having current system length and big endian byte order.

`float_le` Float having current system length and little endian byte order.

`double` Double having current system length and byte order.

`double_be`
: Double having current system length and big endian byte order.

`double_le`
: Double having current system length and little endian byte order.

`bcd_be_`*`len`*
: Bcd number having length *len* and nybbles in big endian order.

`bcd_le_`*`len`*
: Bcd number having length *len* and nybbles in little endian order.

`hex_be_`*`len`*
: Hexadecimal data in big endian order having length *len*.

`hex_le_`*`len`*
: Hexadecimal data in little endian order having length *len*.

If *length* is given instead of the *type*, then the field is assumed to be a printable string having length *length*. String is printed until *length* characters are printed or NULL character is found.

Bcd number (`bcd_be_`*`len`* and `bcd_le_`*`len`*) is printed until *len* bytes are read or a nybble having hexadecimal value `f` is found. Bcd number having big endian order is printed in order: most significant nybble first and least significant nybble second and bcd number having little endian order is printed in order: least significant nybble first and most significant nybble second. Bytes are always read in big endian order.

Hexadecimal data (`hex_be_`*`len`* and `hex_le_`*`len`*) is printed as hexadecimal values. Big endian data is printed starting from lower address and little endian data starting from upper address.

`field-count` *number*
: Same effect as having `"field *"` *number* times. This can be used in separated structure instead of writing sequential `"field *"` definitions. Several `field-count`s can be used in the same record and they can be mixed with `field`.

`fields-from` *record*
: Fields in this record are the same as in record *record*. `field` and `fields-from` are mutually exclusive.

`output` *name*`|no|raw`
: This record is printed according to output format *name*. Default is to use output format specified in structure.

`level` *number* `[`*element_name*`|*`]` `[`*group_name*`]`
: Levels can be used to print the file in hierarchical multi-level nested form document. *number* is the level of the record, starting from number one (highest level), *element_name* is the name for the record, *group_name* is used to group records in the same and lower levels. Only *number* is mandatory. Use * instead of the element name if group name is needed.

`record-length strict|minimum`

`strict`
: Input record length (fixed format) or field count (separated format) must match the record definition in order to get it processed. This is the default value.

`minimum`
: Input record length or field count can be the same or longer as defined for the record. The rest of the input line is ignored.

`variable-length` *record_length variable_length_field adjust*
: *record_length* and *variable_length_field* are the names of two fields in the record and *adjust* is a signed integer. Record length is read from field *record_length*. *record_length* is assumed to be an integer type for binary structures or contain only decimal numbers in fixed length structure. *record_length* is assumed to contain the total length of the record. *variable_length_field* is the field having variable length. The length of *variable_length_field* is calculated by subtracting the total length of the all other fields from the length read from *record_length*. The length given by keyword *field* for *variable_length_field* is ignored. After calculating the length it is adjusted by *adjust*. *adjust* can be used in cases where the length read from *variable_length_field* does not contain the total length of the record. variable-length can be used with binary or fixed lengths structures only.

Output

Keyword `output` specifies a output format for formatting the input data for output. Formatting is controlled using options and printf style directives. An output definition is independent from structure, so one output format can be used with different input file formats.

The output notation:

```
output name {
    option value ...
    ...
}
```

Actual formatting and printing is controlled using *pictures* in output options. Pictures can contain following printf style directives:

`%f` Name of the input file.

`%s` Name of the current structure.

`%r` Name of the current record.

`%o` Input record number in current file.

`%O` Input record number starting from the first file.

`%i` Byte offset of the current record in the current file. Starts from zero.

`%I` Byte offset of the current record starting from the first file. Starts from zero.

`%n` Field name.

`%t` Field contents, without leading and trailing white-spaces.

`%d` Field contents. Binary integer is printed as a decimal value. Floating point number is printed in the style `[-]ddd.ddd`, where the number of digits after the decimal-point character is 6. Bcd number is printed as a decimal number and hexadecimal data as consecutive hexadecimal values.

`%D` Field contents, right padded to the field length (requires length definition for the field).

`%C` Field contents, right padded to the field length (requires length definition for the field). Contents is cut if the input field is longer than output length.

`%x` Unsigned hexadecimal value of a binary integer. Other fields are printed as directive `%d` would be used.

`%l` Lookup value which has been found using current field as a search key.

`%L` Lookup value, right padded to the field length.

`%p` Fields start position in a record. For fixed and binary structure this is field's byte position in the input line and for separated structure this is the ordinal number of the field. Starts from one.

`%h` Hexadecimal dump of a field. Byte values are printed as consecutive `xnn` values, where the `nn` is the hexadecimal value of a byte. Data is printed before any endian conversion.

`%e` Does not print anything, causes still the "field empty" check to be performed. Can be used when only the names of non-empty fields should be printed.

`%g` Group name given by the keyword `group_name` in record definition.

`%m` Element name given by the keyword `element_name` in record definition.

`%%` Percent sign.

Output options:

`file_header` *picture*
: *picture* is printed once before file contents.

`file_trailer` *picture*
: *picture* is printed once after file contents.

`header` *picture*
: If given, then the header line describing the field names is printed before records. Every field name is printed according the *picture* using the same separator and field length as given for the fields. Picture can contain only `%n` directive.

`data` *picture*
: Field contents is printed according *picture*.

`lookup` *picture*
: If current field is related to lookup table, then this *picture* is used instead of picture from `data`. This makes possible to use different picture when the field is related to a lookup table. Default is to use the picture from `data`.

`separator` *string*
: All fields are terminated by *string*, except the last field of the record. Default is not to print separator.

`record_header` *picture*
: *picture* is printed before the record content. Default is not to print the record header.

`record_trailer` *picture*
: *picture* is printed after the record content. Default is newline.

`justify left|right|`*char*
: The output from the `data` option is left or right justified. *char* justifies output according the first occurrence of *char* in the data picture. Default is left.

`indent` *string*
: Record contents is intended by *string*. Field contents is intended by two times the string. Default is not to indent. If file contents is printed in hierarchical form (keyword `level` in record definition) then contents is indented according the level of a record.

`field-list` *name1*`,`*name2*`,...`
: Only fields and constants named as *name1*,*name2*,. . . are printed, same effect as has option `-f`. Default is print all fields and no constants. Fields and constants are also printed in the same order as they are listed.

`no-data-print yes|no`
: If `field-list` is given and and this is set as no and none of the fields in `field-list` does not belong to the current record, then the `record_header` and `record_trailer` are not printed. Default is yes.

`field-empty-print yes|no`
: When set as no, nothing is printed for the fields which consist entirely of characters from `empty-chars`. If none of the fields of a record are printed, then the printing of `record_trailer` is also suppressed. Default is yes.

`empty-chars` *string*
: *string* specifies a set of characters which consist an "empty" field. Default is " \f\n\r\t\v" (space, form-feed, newline, carriage return, horizontal tab and vertical tab).

`output-file` *file*
: Output is written to *file* instead of the default output (standard output or given by `-o, --output`). If - is given the output is written to standard output.

`group_header` *picture*
: If a record has a level and a group name defined, *picture* is printed before the first record in a group or if the group name has changed in the same level. **Note**: Level related pictures can contain printing directives `%g` and `%n` only.

`group_trailer` *picture*
: If a record has a level and a group name defined, *picture* is printed after the records in lower levels are printed or if the group name has changed in the same level or if a higher level record is found.

`element_header` *picture*
: If a record has a level and a element name defined, *picture* is printed before the records contents.

`element_trailer` *picture*
: If a record has a level and a element name defined, *picture* is printed after the records contents or after the following lower level records.

`hex-caps yes|no`
: Print hexadecimal numbers in capital letters. Default is no.

Lookup

Keyword `lookup` specifies a lookup table which can be searched using field contents. Found values can be printed using output directives `%l` and `%L`.

The lookup table notation:

```
lookup name {
    option value ...
    ...
}
```

Lookup options:

`search exact | longest`
: Search method for this table. Either exact or longest match is used when searching the table. Default is `exact`.

`pair` *`key value`*
: Defines a key/value pair for the lookup table. In case of binary file *key* must have the same representation as can be shown using the `%d` printing directive.

`file` *`name`* [*`separator`*]
: Data for the lookup table is read from file *name*. Each line in file *name* is considered as a key/value pair separated by a single character *separator*. Default separator is semicolon. Lines without separator are silently omitted. **Note**: The file size is limited by available memory because the file contents is loaded to memory.

`default-value` *`value`*
: If searching the lookup table is unsuccessful then *value* is used in printing. Default is empty string.

Constants

Keyword `const` specifies one name/value pair which can be used as an additional output field. Constants can be used only in field lists (option `-f,--field-list`, or output option `field-list`).

Constants can be used to add fields to output which do not appear in input. E.g. new fields for separated output or adding spaces after a fixed length field (changing the field length).

Note that *value* is printed as it is for every record. It cannot be changed record by record.

If a constant has the same name as one of the input fields, the value *value* is printed instead of the input field contents.

The constant notation:

```
const name value
```

When *name* appears in field list it is treated as one of the input fields having contents *value*.

Filter

Keyword `filter` defines a command that can be used to format field raw contents. Command must read the standard input and write to standard output and it must not block. Field raw contents is filtered through the command and the output is printed as field contents.

The filter notation:

```
filter name command
```

name is referred in field definition. *command* is the shell command to be executed.

Anonymization

Keyword `anonymize` defines a set of fields which will be anonymized by using command line option `-A,--anonymize` is given. Ffe uses non-reversible anonymization methods and preserves the original field length.

Notation:

```
anonymize name {
    method ...
    ...
}
```

The anonymization will be done if command line option `-A,--anonymize` is given with *name*. Anonymize options:

`method` *field method start length parameter*

All fields named as *field* in the current structure will be anonymized using method *method*. As default the whole field is anonymized. Some parts of the field can be left non-anonymized using *start* and *length*. *start* is the byte position where the anonymization starts, first byte is number 1. If *start* is negative the anonymization starts from the end of the field. If *length* is given then *length* number of bytes is anonymized after start position, default value 0 means the rest of the field. Only *field* and *method* are mandatory.

Values for *method*:

`MASK` Field will be masked with character '0'. Different character can be given with *parameter*.

`RANDOM`
`NRANDOM` Field will be filled with randomly selected bytes.

`HASH`
`NHASH` Field will be filled with data from hash calculated from the original field. This method yields always the same result with same input. The hash length in bytes can be given with *parameter*. Default hash length is 16, valid values for hash length are 16, 32 and 64.

Methods RANDOM and HASH use characters `0-9,A-Z,a-z` and space for text fields. Methods NRANDOM and NHASH use only characters `0-9`. For binary fields all byte values are used. BCD coded fields are always filled with BCD values `0-9`.

Command Substitution

Command Substitution allows the output of a command to replace parts of the configuration file. Syntax for command substitution is:

‘`command`‘

The `command` is executed and the ‘`command`‘ is substituted with the standard output of the command, with any trailing newlines deleted. Command substitutions may not be nested.

Before executing the `command` `ffe` sets following environment variables:

`FFE_STRUCTURE`

The name of the structure from `-s,--structure`.

`FFE_OUTPUT`
: The name of the output file from `-o,--output`.

`FFE_FORMAT`
: The name of the output format from `-p,--print`.

`FFE_FIRST_FILE`
: The name of the first input file.

`FFE_FILES`
: A space-separated list of all input files.

If variable is already set it will not be replaced.

Input Preprocessor

It is possible to define an input preprosessor for `ffe`. An input preprocessor is simply an executable program which writes the contents of the input file to standard output which will be read by `ffe`. If the input preprosessor does not write any characters on its standard output, then `ffe` uses the original file.

To set up an input preprocessor, set the `FFEOPEN` environment variable to a command line which will invoke your input preprocessor. This command line should include one occurrence of the string `%s`, which will be replaced by the input filename when the input preprocessor command is invoked.

The input preprocessor is not used if `ffe` is reading standard input.

Convenient way is to use `lesspipe` (or `lesspipe.sh`), which is available in many UNIX-systems, for example

```
export FFEOPEN="/usr/bin/lesspipe %s"
```

Using the example above is it possible to give a zipped input file to `ffe`, then the input processor will unzip the file before it is processed by `ffe`.

3.3 Guessing

If `-s` is not given, `ffe` tries to guess the input structure.

When guessing binary data `ffe` reads the first block of input data and tries to match the structure definitions from configuration file to that block. The input block size is the maximum binary block size found in configuration file.

When guessing text data `ffe` reads the first 10 000 lines or 1 MB of input data and tries to match the structure definitions from configuration file to input stream. If all lines match one and only one structure, the structure is used for reading the input file.

Guessing uses following execution cycle:

1. A input line or a binary block is read
2. All record `id`'s are compared to the input data, if all `id`'s of a record match the input date and the records line length matches the total length (or total count for separated structure) of the fields, the record is considered to match the input line. If there are no `id`'s, only the line length or field count is checked. In case of binary data only `id`'s are used in matching.

3. In case of text data: If all lines match at least one of the records in a particular structure, the structure is considered as selected. There must be only one structure matching all lines used for guessing.

   In case of binary data: If the first block matches at least one record of a structure, the structure is considered as selected. Only one structure must match.

3.4 Limitations

At least in GNU/Linux `ffe` should be able to handle big files (> 4 GB), other systems are not tested.

Regular expression can be used in operator `?` in option `-e`, `--expression` and in record key word `rid` only in systems where regular expression functions (regcomp, regexec, ...) are available.

4 How ffe works

Following examples use two different input files:

Fixed length example

Fixed length personnel file with header and trailer, line (record) is identified by the first byte (H = Header, E = Employee, B = Boss, T = trailer).

```
$cat personnel.fix
H2006-02-25
EJohn     Ripper       23
BScott    Tiger        45
EMary     Moore        41
ERidge    Forrester    31
T0004
$
```

Structure for reading file above. Note that record '`boss`' reuses fields from '`employee`'.

```
structure personel_fix {
    type fixed
    record header {
        id 1 H
        field type 1
        field date 10
    }
    record employee {
        id 1 E
        field EmpType 1
        field FirstName 9
        field LastName  13
        field Age 2
    }
    record boss {
        id 1 B
        fields-from employee
    }
    record trailer {
        id 1 T
        field type 1
        field count 4
    }
}
```

Separated example

Same file as above, but now separated by comma.

```
$cat personnel.sep
H,2006-02-25
```

```
E,john,Ripper,23
B,Scott,Tiger,45
E,Mary,Moore,41
E,Ridge,Forrester,31
T,0004
$
```

Structure for reading file above. Note that the field lengths are not needed in separated format. Length is need if the separated data is to be printed in fixed length format.

```
structure personel_sep {
    type separated ,
    record header {
        id 1 H
        field type
        field date
    }
    record employee {
        id 1 E
        field type
        field FirstName
        field LastName
        field Age
    }
    record boss {
        id 1 B
        fields-from employee
    }
    record trailer {
        id 1 T
        field type
        field count
    }
}
```

Printing in XML format

Data in examples above can be printed in XML using output definition like:

```
output xml {
    file_header "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
    data "<%n>%t</%n>\n"
    record_header "<%r>\n"
    record_trailer "</%r>\n"
    indent " "
}
```

Example output using command (assuming definitions above are saved in ~/.fferc)

```
ffe -p xml personnel.sep
```

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<header>
 <type>H</type>
 <date>2006-02-25</date>
</header>
<employee>
 <type>E</type>
 <FirstName>john</FirstName>
 <LastName>Ripper</LastName>
 <Age>23</Age>
</employee>
<boss>
 <type>B</type>
 <FirstName>Scott</FirstName>
 <LastName>Tiger</LastName>
 <Age>45</Age>
</boss>
<employee>
 <type>E</type>
 <FirstName>Mary</FirstName>
 <LastName>Moore</LastName>
 <Age>41</Age>
</employee>
<employee>
 <type>E</type>
 <FirstName>Ridge</FirstName>
 <LastName>Forrester</LastName>
 <Age>31</Age>
</employee>
<trailer>
 <type>T</type>
 <count>0004</count>
</trailer>
```

Printing sql commands

Data in examples above can be loaded to database by generated sql commands. Note that the header and trailer are not loaded, because only fields '`FirstName`','`LastName`' and '`Age`' are printed and '`no-data-print`' is set as no. This prevents the '`record_header`' and '`record_trailer`' to be printed for file header and trailer.

```
output sql {
    file_header "delete table boss;\ndelete table employee;\n"
    record_header "insert into %r values("
    data "'%t'"
    separator ","
    record_trailer ");\n"
    file_trailer "commit\nquit\n"
    no-data-print no
```

```
        field-list FirstName,LastName,Age
    }
```

Output from command

```
ffe -p sql personnel.sep
```

```
delete table boss;
delete table employee;
insert into employee values('john','Ripper','23');
insert into boss values('Scott','Tiger','45');
insert into employee values('Mary','Moore','41');
insert into employee values('Ridge','Forrester','31');
commit
quit
```

Human readable output

This output format shows the fields in format suitable for displaying in screen or printing.

```
output nice {
    record_header "%s - %r - %f - %o\n"
    data "%n=%t\n"
    justify =
    indent " "
}
```

Output from command

```
ffe -p nice personnel.fix
```

```
personel - header - personnel.fix - 1
 type=H
 date=2006-02-25

personel - employee - personnel.fix - 2
   EmpType=E
 FirstName=John
  LastName=Ripper
       Age=23

personel - boss - personnel.fix - 3
   EmpType=B
 FirstName=Scott
  LastName=Tiger
       Age=45

personel - employee - personnel.fix - 4
   EmpType=E
 FirstName=Mary
  LastName=Moore
       Age=41
```

```
personel - employee - personnel.fix - 5
   EmpType=E
 FirstName=Ridge
  LastName=Forrester
       Age=31

personel - trailer - personnel.fix - 6
  type=T
 count=0004
```

HTML table

Personnel data can be displayed as HTML table using output like:

```
output html {
    file_header "<html>\n<head>\n</head>\n<body>\n<table border=\"1\">\n<tr>\n"
    header "<th>%n</th>\n"
    record_header "<tr>\n"
    data "<td>%t</td>\n"
    file_trailer "</table>\n</body>\n</html>\n"
    no-data-print no
}
```

Output from command

```
ffe -p html -f FirstName,LastName,Age personnel.fix
```

```
<html>
<head>
</head>
<body>
<table border="1">
<tr>
<th>FirstName</th>
<th>LastName</th>
<th>Age</th>

<tr>
<td>John</td>
<td>Ripper</td>
<td>23</td>

<tr>
<td>Scott</td>
<td>Tiger</td>
<td>45</td>

<tr>
<td>Mary</td>
<td>Moore</td>
<td>41</td>
```

```
<tr>
<td>Ridge</td>
<td>Forrester</td>
<td>31</td>

</table>
</body>
</html>
```

Using expression

Printing only Scott's record using expression with previous example:

```
ffe -p html -f FirstName,LastName,Age -e FirstNameˆScott personnel.fix
```

```
<html>
<head>
</head>
<body>
<table border="1">
<tr>
<th>FirstName</th>
<th>LastName</th>
<th>Age</th>

<tr>
<td>Scott</td>
<td>Tiger</td>
<td>45</td>

</table>
</body>
</html>
```

Using replace

Make all bosses and write a new personnel file printing the fields in fixed length format using directive %D:

Output definition:

```
output fixed
{
    data "%D"
}
```

Write a new file:

```
$ffe -p fixed -r EmpType=B -o personnel.fix.new personnel.fix
$cat personnel.fix.new
H2006-02-25
BJohn     Ripper       23
```

```
BScott    Tiger        45
BMary     Moore        41
BRidge    Forrester    31
T0004
$
```

Using constant

The length of the fields FirstName and LastName in fixed length format will be made two bytes longer. This will be done by printing a constant after those two fields. We use dots instead of spaces in order to make change more visible.

Because we do not want to change header and trailer we need specially crafted configuration file. Employee and boss records will be printed using new output *fixed2* and other records will be printed using output *default*.

New definition file `new_fixed.rc`:

```
const 2dots ".."

structure personel_fix {
    type fixed
    record header {
        id 1 H
        field type 1
        field date 10
    }
    record employee {
        id 1 E
        field EmpType 1
        field FirstName 9
        field LastName  13
        field Age 2
        output fixed2
    }
    record boss {
        id 1 B
        fields-from employee
        output fixed2
    }
    record trailer {
        id 1 T
        field type 1
        field count 4
    }
}

output default
{
    data "%D"
```

```
}

output fixed2
{
    data "%D"
    field-list Emptype,FirstName,2dots,LastName,2dots,Age
}
```

Print new flat file:

```
$ ffe -c new_fixed.rc personel_fix
H2006-02-25
EJohn     ..Ripper       ..23
BScott    ..Tiger        ..45
EMary     ..Moore        ..41
ERidge    ..Forrester    ..31
T0004
$
```

Using lookup table

Lookup table is used to explain the EmpTypes contents in output format `nice`:

Lookup definition:

```
lookup Type
{
    search exact
    pair H Header
    pair B "He is a Boss!"
    pair E "Not a Boss!"
    pair T Trailer
    default-value "Unknown record type!"
}
```

Mapping the EmpType field to lookup:

```
structure personel_fix {
    type fixed
    record header {
        id 1 H
        field type 1
        field date 10
    }
    record employee {
        id 1 E
        field EmpType 1 Type
        field FirstName 9
        field LastName  13
        field Age 2
    }
    record boss {
```

```
        id 1 B
        fields-from employee
    }
    record trailer {
        id 1 T
        field type 1
        field count 4
    }
}
```

Adding the lookup option to output definition nice.

```
output nice {
    record_header "%s - %r - %f - %o\n"
    data "%n=%t\n"
    lookup "%n=%t (%l)\n"
    justify =
    indent " "
}
```

Running ffe:

```
$ffe -p nice personnel.fix
personel_fix - header - personel_fix - 1
 type=H
 date=2006-02-25

personel_fix - employee - personel_fix - 2
   EmpType=E (Not a Boss!)
 FirstName=John
  LastName=Ripper
       Age=23

personel_fix - boss - personel_fix - 3
   EmpType=B (He is a Boss!)
 FirstName=Scott
  LastName=Tiger
       Age=45

personel_fix - employee - personel_fix - 4
   EmpType=E (Not a Boss!)
 FirstName=Mary
  LastName=Moore
       Age=41

personel_fix - employee - personel_fix - 5
   EmpType=E (Not a Boss!)
 FirstName=Ridge
  LastName=Forrester
       Age=31
```

```
personel_fix - trailer - personel_fix - 6
  type=T
 count=0004
```

External lookup file

In previous example the lookup data could be read from external file like:

```
$cat lookupdata
H;Header
B;He is a Boss!
E;Not a Boss!
T;Trailer
$
```

Lookup definition using file above:

```
lookup Type
{
    search exact
    file lookupdata
    default-value "Unknown record type!"
}
```

Making universal csv reader using command substitution

Command substitution can be used to make a configuration for reading any csv file. The number of fields will be read from the first file using awk. Input file names and date are printed in the file header:

```
structure csv {
    type separated ,
    header first
    record csv {
        field-count ‘awk "-F," ’FNR == 1 {print NF;exit;}’ $FFE_FIRST_FILE‘
    }
}

output default {
    file_header "Files: ‘echo $FFE_FILES‘\n‘date‘\n"
    data "%n=%d\n"
    justify =
}
```

Reading binary data

A binary block having a 3 byte text (ABC) in 5 bytes long space, one byte integer (35), a 32 bit integer (12345678), a double (345.385), a 3 byte bcd number (45112) and a 4 byte hexadecimal data (f15a9188) can be read using following configuration:

```
structure bin_data
{
```

```
        type binary
        record b
        {
            field text 5
            field byte_int int8
            field integer int
            field number double
            field bcd_number bcd_be_3
            field hex hex_be_4
        }
    }

    output default
    {
        data "%n = %d (%h)\n"
    }
```

The %h directive gives a hex dump of the input data.

Hexadecimal dump of the data:

```
$ od -t x1 example_bin
0000000 41 42 43 00 08 23 4e 61 bc 00 5c 8f c2 f5 28 96
0000020 75 40 45 11 2f f1 5a 91 88
0000031
```

Using ffe:

```
$ffe -c example_bin.fferc -s bin_data example_bin
text = ABC (x41x42x43x00x08)
byte_int = 35 (x23)
integer = 12345678 (x4ex61xbcx00)
number = 345.385000 (x5cx8fxc2xf5x28x96x75x40)
bcd_number = 45112 (x45x11x2f)
hex = f15a9188 (xf1x5ax91x88)
```

Note that the text has only 3 characters before NULL byte. Because this example was made in little endian machine, same result can be achieved with different configuration:

```
structure bin_data
{
    type binary
    record b
    {
        field text 5
        field byte_int int8
        field integer int32_le
        field number double_le
        field bcd_number bcd_be_3
        field hex hex_be_4
    }
}
```

This configuration is more portable in case the same data is to be read in a different architecture because endianess of integer and double are explicit given.

If the bcd number is read with `bcd_le_3` it would look as

```
bcd_number = 5411 (x45x11x2f)
```

Note that nybbles are swapped and last byte is handled as `f2` (`f` stops the printing) causing only first two bytes to be printed.

and if hexadecimal data is read with `hex_le_4` it would look as

```
hex = 88915af1 (xf1x5ax91x88)
```

Bytes are printed starting from the end of the data.

Printing nested XML

The keyword `level` in record definition can be used to print data in multi-level nested form. In this example a parent row is in level one and a child row is in level two. Children after a parent row belongs to the parent before child rows, so they are enclosed in a parent element.

Example data:

```
P,John Smith,3
C,Kathren,6,Blue
C,Jimmy,4,Red
C,Peter,2,Green
P,Margaret Eelers,2
C,Aden,16,White
C,Amanda,20,Black
```

A parent row consists of ID (P), parent name, and the count of the children. A child row consists of id (C), child name, age and favorite color.

This can be printed in nested XML using rc file:

```
structure family
{
    type separated ,
    record parent
    {
        id 1 P
        field FILLER
        field Name
        field Child_count
        level 1 parent
    }

    record child
    {
        id 1 C
        field FILLER
        field Name
        field Age
        field FavoriteColor
```

```
        level 2 child children
    }
}

output nested_xml
{
    file_header "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
    data "<%n>%t</%n>\n"
    indent " "
    record_trailer ""
    group_header "<%g>\n"
    group_trailer "</%g>\n"
    element_header "<%m>\n"
    element_trailer "</%m>\n"
}
```

Output:

```
<?xml version="1.0" encoding="UTF-8"?>
 <parent>
  <Name>John Smith</Name>
  <Child_count>3</Child_count>
  <children>
   <child>
    <Name>Kathren</Name>
    <Age>6</Age>
    <FavoriteColor>Blue</FavoriteColor>
   </child>
   <child>
    <Name>Jimmy</Name>
    <Age>4</Age>
    <FavoriteColor>Red</FavoriteColor>
   </child>
   <child>
    <Name>Peter</Name>
    <Age>2</Age>
    <FavoriteColor>Green</FavoriteColor>
   </child>
  </children>
 </parent>
 <parent>
  <Name>Margaret Eelers</Name>
  <Child_count>2</Child_count>
  <children>
   <child>
    <Name>Aden</Name>
    <Age>16</Age>
    <FavoriteColor>White</FavoriteColor>
```

```
   </child>
   <child>
    <Name>Amanda</Name>
    <Age>20</Age>
    <FavoriteColor>Black</FavoriteColor>
   </child>
  </children>
 </parent>
```

Some examples put in a single file

```
structure personel_fix {
    type fixed
    record header {
        id 1 H
        field type 1
        field date 10
    }
    record employee {
        id 1 E
        field EmpType 1 Type
        field FirstName 9
        field LastName  13
        field Age 2
    }
    record boss {
        id 1 B
        fields-from employee
    }
    record trailer {
        id 1 T
        field type 1
        field count 4
    }
}

structure personel_sep {
    type separated ,
    record header {
        id 1 H
        field type
        field date
    }
    record employee {
        id 1 E
        field type
        field FirstName
```

```
        field LastName
        field Age
    }
    record boss {
        id 1 B
        fields-from employee
    }
        record trailer {
        id 1 T
        field type
        field count
    }
}

structure bin_data
{
    type binary
    record b
    {
        field text 5
        field byte_int int8
        field integer int32_le
        field number double_le
        field bcd_number bcd_be_3
        field hex hex_be_4
    }
}

output xml {
    file_header "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
    data "<%n>%t</%n>\n"
    record_header "<%r>\n"
    record_trailer "</%r>\n"
    indent " "
}

output sql {
    file_header "delete table boss;\ndelete table employee;\n"
    record_header "insert into %r values("
    data "'%t'"
    separator ","
    record_trailer ");\n"
    file_trailer "commit\nquit\n"
    no-data-print no
    field-list FirstName,LastName,Age
}
```

```
output nice {
    record_header "%s - %r - %f - %o\n"
    data "%n=%t\n"
    lookup "%n=%t (%l)\n"
    justify =
    indent " "
}

output html {
    file_header "<html>\n<head>\n</head>\n<body>\n<table border=\"1\">\n<tr>\n"
    header "<th>%n</th>\n"
    record_header "<tr>\n"
    data "<td>%t</td>\n"
    file_trailer "</table>\n</body>\n</html>\n"
    no-data-print no
}

output fixed
{
    data "%D"
}

lookup Type
{
    search exact
    pair H Header
    pair B "He is a Boss!"
    pair E "Not a Boss!"
    pair T Trailer
    default-value "Unknown record type!"
}
```

Anonymization

Anonymize fields FirstName, LastName and Age for personnel data:

```
anonymize personnel
{
    method FirstName HASH 2
    method LastName HASH 2
    method Age NRANDOM
}
```

Data before anonymization:

```
$cat personnel.fix
H2006-02-25
EJohn     Ripper       23
BScott    Tiger        45
EMary     Moore        41
```

```
ERidge    Forrester    31
T0004
```

Anonymize the data to new file `personnel_anon.fix` (using the default configuration file `~/.fferc` and raw output):

```
ffe -A personnel -praw -o personnel_anon.fix personnel.fix
```

Anonymized data:

```
$cat personnel_anon.fix
H2006-02-25
EJQIQ9C5oBR2rDU0qiSTv7E62
BSqUcsYzSTTNTuTraspsG4154
EMTsXkHltVMsV8qmK1tkgq 00
ER1e90zv1dFjP4 xgflVGQF87
T0004

$ffe -pnice personnel_anon.fix
 personel - header - personnel_anon.fix - 1
   type=H
   date=2006-02-25

 personel - employee - personnel_anon.fix - 2
     EmpType=E
   FirstName=JQIQ9C5oB
    LastName=R2rDU0qiSTv7E
         Age=62

 personel - boss - personnel_anon.fix - 3
    EmpType=B
  FirstName=SqUcsYzST
   LastName=TNTuTraspsG41
        Age=54

 personel - employee - personnel_anon.fix - 4
    EmpType=E
  FirstName=MTsXkHltV
   LastName=MsV8qmK1tkgq
        Age=00

 personel - employee - personnel_anon.fix - 5
    EmpType=E
  FirstName=R1e90zv1d
   LastName=FjP4 xgflVGQF
        Age=87

 personel - trailer - personnel_anon.fix - 6
   type=T
  count=0004
```

FirstName and LastName have preserved the first letter because anonymization started from the second byte. Age is a two digit random number. Name fields will get the same anonymized value for each run, but Age will have a random value for each run.

Using ffe to test file integrity

ffe can be used to check flat file integrity, because ffe checks for all lines the line length and id's for fixed length structure and field count and id's for separated structure.

Integrity can be checked using command

```
ffe -p no -l inputfiles...
```

Because option -p has value no nothing is printed to output except the error messages. Option -l causes all erroneous lines to be reported, not just the first one.

Example output:

```
ffe: Invalid input line in file 'inputfileB', line 14550
ffe: Invalid input line in file 'inputfileD', line 12
```

5 Reporting Bugs

If you find a bug in `ffe`, please send electronic mail to `tjsa@iki.fi`. Include the version number, which you can find by running '`ffe --version`'. Also include in your message the output that the program produced and the output you expected.

If you have other questions, comments or suggestions about `ffe`, contact the author via electronic mail to `tjsa@iki.fi`. The author will try to help you out, although he may not have time to fix your problems.

Table of Contents