Forms Mode User's Manual

Forms-Mode version 2

for GNU Emacs 20.1

June 1997

Johan Vromans
jvromans@squirrel.nl

1 Forms Example

Let's illustrate Forms mode with an example. Suppose you are looking at the '`/etc/passwd`' file, and the screen looks like this:

```
====== /etc/passwd ======

User : root   Uid: 0   Gid: 1

Name : Super User

Home : /

Shell: /bin/sh
```

As you can see, the familiar fields from the entry for the super user are all there, but instead of being colon-separated on one single line, they make up a forms.

The contents of the forms consist of the contents of the fields of the record (e.g. '`root`', '`0`', '`1`', '`Super User`') interspersed with normal text (e.g '`User : `', '`Uid: `').

If you modify the contents of the fields, Forms mode will analyze your changes and update the file appropriately. You cannot modify the interspersed explanatory text (unless you go to some trouble about it), because that is marked read-only (see section "Text Properties" in *The Emacs Lisp Reference Manual*).

The Forms mode control file specifies the relationship between the format of '`/etc/passwd`' and what appears on the screen in Forms mode. See Chapter 5 [Control File Format], page 9.

2 Entering and Exiting Forms Mode

M-x forms-find-file ⟨RET⟩ *control-file* ⟨RET⟩
: Visit a database using Forms mode. Specify the name of the **control file**, not the data file!

M-x forms-find-file-other-window ⟨RET⟩ *control-file* ⟨RET⟩
: Similar, but displays the file in another window.

The command `forms-find-file` evaluates the file *control-file*, and also visits it in Forms mode. What you see in its buffer is not the contents of this file, but rather a single record of the corresponding data file that is visited in its own buffer. So there are two buffers involved in Forms mode: the *forms buffer* that is initially used to visit the control file and that shows the records being browsed, and the *data buffer* that holds the data file being visited. The latter buffer is normally not visible.

Initially, the first record is displayed in the forms buffer. The mode line displays the major mode name '`Forms`', followed by the minor mode '`View`' if the data base is read-only. The number of the current record (*n*) and the total number of records in the file(*t*) are shown in the mode line as '*n*/*t*'. For example:

```
--%%-Emacs: passwd-demo          (Forms View 1/54)----All-------
```

If the buffer is not read-only, you may change the buffer to modify the fields in the record. When you move to a different record, the contents of the buffer are parsed using the specifications in `forms-format-list`, and the data file is updated. If the record has fields that aren't included in the display, they are not changed.

Entering Forms mode runs the normal hook `forms-mode-hooks` to perform user-defined customization.

To save any modified data, you can use *C-x C-s* (`forms-save-buffer`). This does not save the forms buffer (which would be rather useless), but instead saves the buffer visiting the data file.

To terminate Forms mode, you can use *C-x C-s* (`forms-save-buffer`) and then kill the forms buffer. However, the data buffer will still remain. If this is not desired, you have to kill this buffer too.

3 Forms Commands

The commands of Forms mode belong to the `C-c` prefix, with one exception: ⟨TAB⟩, which moves to the next field. Forms mode uses different key maps for normal mode and read-only mode. In read-only Forms mode, you can access most of the commands without the `C-c` prefix, but you must type ordinary letters instead of control characters; for example, type `n` instead of `C-c C-n`.

If your Emacs has been built with X-toolkit support, Forms mode will provide its own menu with a number of Forms mode commands.

`C-c C-n`
: Show the next record (`forms-next-record`). With a numeric argument *n*, show the *n*th next record.

`C-c C-p`
: Show the previous record (`forms-prev-record`). With a numeric argument *n*, show the *n*th previous record.

`C-c C-l`
: Jump to a record by number (`forms-jump-record`). Specify the record number with a numeric argument.

`C-c <`
: Jump to the first record (`forms-first-record`).

`C-c >`
: Jump to the last record (`forms-last-record`). This command also recalculates the number of records in the data file.

⟨TAB⟩
`C-c` ⟨TAB⟩
: Jump to the next field in the current record (`forms-next-field`). With a numeric argument *n*, jump forward *n* fields. If this command would move past the last field, it wraps around to the first field.

`C-c C-q`
: Toggles read-only mode (`forms-toggle-read-only`). In read-only Forms mode, you cannot edit the fields; most Forms mode commands can be accessed without the prefix `C-c` if you use the normal letter instead (for example, type `n` instead of `C-c C-n`). In edit mode, you can edit the fields and thus change the contents of the data base; you must begin Forms mode commands with `C-c`. Switching to edit mode is allowed only if you have write access to the data file.

`C-c C-o`
: Create a new record and insert it before the current record (`forms-insert-record`). It starts out with empty (or default) contents for its fields; you can then edit the fields. With a numeric argument, the new record is created *after* the current one. See also `forms-modified-record-filter` in Chapter 7 [Modifying Forms Contents], page 13.

`C-c C-k`
: Delete the current record (`forms-delete-record`). You are prompted for confirmation before the record is deleted unless a numeric argument has been provided.

`C-c C-s` *regexp* ⟨RET⟩
: Search forward for *regexp* in all records following this one (`forms-search-forward`). If found, this record is shown. If you give an empty argument, the previous regexp is used again.

C-c C-r *regexp* ⟨RET⟩
: Search backward for *regexp* in all records following this one (`forms-search-backward`). If found, this record is shown. If you give an empty argument, the previous regexp is used again.

M-x forms-prev-field
: Similar to `forms-next-field` but moves backwards.

M-x forms-save-buffer
C-x C-s
: Forms mode replacement for `save-buffer`. When executed in the forms buffer it will save the contents of the (modified) data buffer instead. In Forms mode this function will be bound to *C-x C-s*.

M-x forms-print
: This command can be used to make a formatted print of the contents of the data file.

In addition the command *M-x revert-buffer* is useful in Forms mode just as in other modes.

The following function key definitions are set up in Forms mode (whether read-only or not):

next	forms-next-record
prior	forms-prev-record
begin	forms-first-record
end	forms-last-record
S-Tab	forms-prev-field

4 Data File Format

Files for use with Forms mode are very simple—each *record* (usually one line) forms the contents of one form. Each record consists of a number of *fields*, which are separated by the value of the string `forms-field-sep`, which is `"\t"` (a Tab) by default.

If the format of the data file is not suitable enough you can define the filter functions `forms-read-file-filter` and `forms-write-file-filter`. `forms-read-file-filter` is called when the data file is read from disk into the data buffer. It operates on the data buffer, ignoring read-only protections. When the data file is saved to disk `forms-write-file-filter` is called to cancel the effects of `forms-read-file-filter`. After being saved, `forms-read-file-filter` is called again to prepare the data buffer for further processing.

Fields may contain text which shows up in the forms in multiple lines. These lines are separated in the field using a “pseudo-newline” character which is defined by the value of the string `forms-multi-line`. Its default value is `"\^k"` (a Control-K character). If it is set to `nil`, multiple line fields are prohibited.

If the data file does not exist, it is automatically created.

5 Control File Format

The Forms mode *control file* serves two purposes. First, it names the data file to use, and defines its format and properties. Second, the Emacs buffer it occupies is used by Forms mode to display the forms.

The contents of the control file are evaluated as a Lisp program. It should set the following Lisp variables to suitable values:

`forms-file`
: This variable specifies the name of the data file. Example:

```
(setq forms-file "my/data-file")
```

If the control file doesn't set `forms-file`, Forms mode reports an error.

`forms-format-list`
: This variable describes the way the fields of the record are formatted on the screen. For details, see Chapter 6 [Format Description], page 11.

`forms-number-of-fields`
: This variable holds the number of fields in each record of the data file. Example:

```
(setq forms-number-of-fields 10)
```

If the control file does not set `forms-format-list` a default format is used. In this situation, Forms mode will deduce the number of fields from the data file providing this file exists and `forms-number-of-records` has not been set in the control file.

The control file can optionally set the following additional Forms mode variables. Most of them have default values that are good for most applications.

`forms-field-sep`
: This variable may be used to designate the string which separates the fields in the records of the data file. If not set, it defaults to the string `"\t"` (a Tab character). Example:

```
(setq forms-field-sep "\t")
```

`forms-read-only`
: If the value is non-`nil`, the data file is treated read-only. (Forms mode also treats the data file as read-only if you don't have access to write it.) Example:

```
(set forms-read-only t)
```

`forms-multi-line`
: This variable specifies the *pseudo newline* separator that allows multi-line fields. This separator goes between the “lines” within a field—thus, the field doesn't really contain multiple lines, but it appears that way when displayed in Forms mode. If the value is `nil`, multi-line text fields are prohibited. The pseudo newline must not be a character contained in `forms-field-sep`.

The default value is `"\^k"`, the character Control-K. Example:

```
(setq forms-multi-line "\^k")
```

`forms-read-file-filter`
: This variable holds the name of a function to be called after the data file has been read in. This can be used to transform the contents of the data file into a format more suitable for forms processing. If it is `nil`, no function is called. For example, to maintain a gzipped database:

```
(defun gzip-read-file-filter ()
  (shell-command-on-region (point-min) (point-max)
                           "gzip -d" t t))
(setq forms-read-file-filter 'gzip-read-file-filter)
```

`forms-write-file-filter`
: This variable holds the name of a function to be called before writing out the contents of the data file. This can be used to undo the effects of `forms-read-file-filter`. If it is `nil`, no function is called. Example:

```
(defun gzip-write-file-filter ()
  (make-variable-buffer-local 'require-final-newline)
  (setq require-final-newline nil)
  (shell-command-on-region (point-min) (point-max)
                           "gzip" t t))
(setq forms-write-file-filter 'gzip-write-file-filter)
```

`forms-new-record-filter`
: This variable holds a function to be called whenever a new record is created to supply default values for fields. If it is `nil`, no function is called. See Chapter 7 [Modifying Forms Contents], page 13, for details.

`forms-modified-record-filter`
: This variable holds a function to be called whenever a record is modified, just before updating the Forms data file. If it is `nil`, no function is called. See Chapter 7 [Modifying Forms Contents], page 13, for details.

`forms-insert-after`
: If this variable is not `nil`, new records are created *after* the current record. Also, upon visiting a file, the initial position will be at the last record instead of the first one.

`forms-check-number-of-fields`
: Normally each record is checked to contain the correct number of fields. Under certain circumstances, this can be undesirable. If this variable is set to `nil`, these checks will be bypassed.

6 The Format Description

The variable `forms-format-list` specifies the format of the data in the data file, and how to convert the data for display in Forms mode. Its value must be a list of Forms mode *formatting elements*, each of which can be a string, a number, a Lisp list, or a Lisp symbol that evaluates to one of those. The formatting elements are processed in the order they appear in the list.

string A string formatting element is inserted in the forms “as is,” as text that the user cannot alter.

number A number element selects a field of the record. The contents of this field are inserted in the display at this point. Field numbers count starting from 1 (one).

list A formatting element that is a list specifies a function call. This function is called every time a record is displayed, and its result, which must be a string, is inserted in the display text. The function should do nothing but returning a string.

The function you call can access the fields of the record as a list in the variable `forms-fields`.

symbol A symbol used as a formatting element should evaluate to a string, number, or list; the value is interpreted as a formatting element, as described above.

If a record does not contain the number of fields as specified in `forms-number-of-fields`, a warning message will be printed. Excess fields are ignored, missing fields are set to empty.

The control file which displays ‘`/etc/passwd`’ file as demonstrated in the beginning of this manual might look as follows:

```
;; This demo visits ‘/etc/passwd’.

(setq forms-file "/etc/passwd")
(setq forms-number-of-fields 7)
(setq forms-read-only t)                 ; to make sure
(setq forms-field-sep ":")
;; Don’t allow multi-line fields.
(setq forms-multi-line nil)

(setq forms-format-list
      (list
       "====== /etc/passwd ======\n\n"
       "User : "    1
       "   Uid: "   3
       "   Gid: "   4
       "\n\n"
       "Name : "    5
       "\n\n"
       "Home : "    6
       "\n\n"
       "Shell: "    7
```

```
           "\n"))
```

When you construct the value of `forms-format-list`, you should usually either quote the whole value, like this,

```
(setq forms-format-list
     '(
       "====== " forms-file " ======\n\n"
       "User : "    1
       (make-string 20 ?-)
       ...
      ))
```

or quote the elements which are lists, like this:

```
(setq forms-format-list
      (list
       "====== " forms-file " ======\n\n"
       "User : "    1
       '(make-string 20 ?-)
       ...
      ))
```

Forms mode validates the contents of `forms-format-list` when you visit a database. If there are errors, processing is aborted with an error message which includes a descriptive text. See Chapter 9 [Error Messages], page 17, for a detailed list of error messages.

If no `forms-format-list` is specified, Forms mode will supply a default format list. This list contains the name of the file being visited, and a simple label for each field indicating the field number.

7 Modifying The Forms Contents

If `forms-read-only` is `nil`, the user can modify the fields and records of the database.

All normal editing commands are available for editing the contents of the displayed record. You cannot delete or modify the fixed, explanatory text that comes from string formatting elements, but you can modify the actual field contents.

If the variable `forms-modified-record-filter` is non-`nil`, it is called as a function before the new data is written to the data file. The function receives one argument, a vector that contains the contents of the fields of the record.

The function can refer to fields with `aref` and modify them with `aset`. The first field has number 1 (one); thus, element 0 of the vector is not used. The function should return the same vector it was passed; the (possibly modified) contents of the vector determine what is actually written in the file. Here is an example:

```
(defun my-modified-record-filter (record)
  ;; Modify second field.
  (aset record 2 (current-time-string))
  ;; Return the field vector.
  record)

(setq forms-modified-record-filter 'my-modified-record-filter)
```

If the variable `forms-new-record-filter` is non-`nil`, its value is a function to be called to fill in default values for the fields of a new record. The function is passed a vector of empty strings, one for each field; it should return the same vector, with the desired field values stored in it. Fields are numbered starting from 1 (one). Example:

```
(defun my-new-record-filter (fields)
  (aset fields 5 (login-name))
  (aset fields 1 (current-time-string))
  fields)

(setq forms-new-record-filter 'my-new-record-filter)
```

8 Miscellaneous

The global variable `forms-version` holds the version information of the Forms mode software.

It is very convenient to use symbolic names for the fields in a record. The function `forms-enumerate` provides an elegant means to define a series of variables whose values are consecutive integers. The function returns the highest number used, so it can be used to set `forms-number-of-fields` also. For example:

```
(setq forms-number-of-fields
      (forms-enumerate
       '(field1 field2 field3 ...)))
```

This sets `field1` to 1, `field2` to 2, and so on.

Care has been taken to keep the Forms mode variables buffer-local, so it is possible to visit multiple files in Forms mode simultaneously, even if they have different properties.

If you have visited the control file in normal fashion with `find-file` or a like command, you can switch to Forms mode with the command `M-x forms-mode`. If you put '`-*- forms -*-`' in the first line of the control file, then visiting it enables Forms mode automatically. But this makes it hard to edit the control file itself, so you'd better think twice before using this.

The default format for the data file, using `"\t"` to separate fields and `"\^k"` to separate lines within a field, matches the file format of some popular database programs, e.g. FileMaker. So `forms-mode` can decrease the need to use proprietary software.

9 Error Messages

This section describes all error messages which can be generated by forms mode. Error messages that result from parsing the control file all start with the text '`Forms control file error`'. Messages generated while analyzing the definition of `forms-format-list` start with '`Forms format error`'.

`Forms control file error: 'forms-file' has not been set`
: The variable `forms-file` was not set by the control file.

`Forms control file error: 'forms-number-of-fields' has not been set`
: The variable `forms-number-of-fields` was not set by the control file.

`Forms control file error: 'forms-number-of-fields' must be a number > 0`
: The variable `forms-number-of-fields` did not contain a positive number.

`Forms control file error: 'forms-field-sep' is not a string`
`Forms control file error: 'forms-multi-line' must be nil or a one-character string`
: The variable `forms-multi-line` was set to something other than `nil` or a single-character string.

`Forms control file error: 'forms-multi-line' is equal to 'forms-field-sep'`
: The variable `forms-multi-line` may not be equal to `forms-field-sep` for this would make it impossible to distinguish fields and the lines in the fields.

`Forms control file error: 'forms-new-record-filter' is not a function`
`Forms control file error: 'forms-modified-record-filter' is not a function`
: The variable has been set to something else than a function.

`Forms control file error: 'forms-format-list' is not a list`
: The variable `forms-format-list` was not set to a Lisp list by the control file.

`Forms format error: field number` *xx* `out of range 1..`*nn*
: A field number was supplied in `forms-format-list` with a value of *xx*, which was not greater than zero and smaller than or equal to the number of fields in the forms, *nn*.

`Forms format error:` *fun* `is not a function`
: The first element of a list which is an element of `forms-format-list` was not a valid Lisp function.

`Forms format error: invalid element` *xx*
: A list element was supplied in `forms-format-list` which was not a string, number or list.

`Warning: this record has` *xx* `fields instead of` *yy*
: The number of fields in this record in the data file did not match `forms-number-of-fields`. Missing fields will be made empty.

`Multi-line fields in this record - update refused!`
: The current record contains newline characters, hence can not be written back to the data file, for it would corrupt it. Probably you inserted a newline in a field, while `forms-multi-line` was `nil`.

`Field separator occurs in record - update refused!`
: The current record contains the field separator string inside one of the fields. It can not be written back to the data file, for it would corrupt it. Probably you inserted the field separator string in a field.

`Record number` *xx* `out of range 1..`*yy*
: A jump was made to non-existing record *xx*. *yy* denotes the number of records in the file.

`Stuck at record` *xx*
: An internal error prevented a specific record from being retrieved.

`No write access to "`*file*`"`
: An attempt was made to enable edit mode on a file that has been write protected.

`Search failed:` *regexp*
: The *regexp* could not be found in the data file. Forward searching is done from the current location until the end of the file, then retrying from the beginning of the file until the current location. Backward searching is done from the current location until the beginning of the file, then retrying from the end of the file until the current location.

`Wrapped` A search completed successfully after wrapping around.

`Warning: number of records changed to` *nn*
: Forms mode's idea of the number of records has been adjusted to the number of records actually present in the data file.

`Problem saving buffers?`
: An error occurred while saving the data file buffer. Most likely, Emacs did ask to confirm deleting the buffer because it had been modified, and you said 'no'.

10 Long Example

The following example exploits most of the features of Forms mode. This example is included in the distribution as file '`forms-d2.el`'.

```
;; demo2 -- demo forms-mode     -*- emacs-lisp -*-

;; This sample forms exploit most of the features of forms mode.

;; Set the name of the data file.
(setq forms-file "forms-d2.dat")

;; Use forms-enumerate to set field names and number thereof.
(setq forms-number-of-fields
      (forms-enumerate
       '(arch-newsgroup                 ; 1
         arch-volume                    ; 2
         arch-issue                     ; and ...
         arch-article                   ; ... so
         arch-shortname                 ; ... ... on
         arch-parts
         arch-from
         arch-longname
         arch-keywords
         arch-date
         arch-remarks)))

;; The following functions are used by this form for layout purposes.
;;
(defun arch-tocol (target &optional fill)
  "Produces a string to skip to column TARGET.
Prepends newline if needed.
The optional FILL should be a character, used to fill to the column."
  (if (null fill)
      (setq fill ? ))
  (if (< target (current-column))
      (concat "\n" (make-string target fill))
    (make-string (- target (current-column)) fill)))
;;
(defun arch-rj (target field &optional fill)
  "Produces a string to skip to column TARGET\
 minus the width of field FIELD.
Prepends newline if needed.
The optional FILL should be a character,
used to fill to the column."
  (arch-tocol (- target (length (nth field forms-fields))) fill))

;; Record filters.
;;
(defun new-record-filter (the-record)
  "Form a new record with some defaults."
  (aset the-record arch-from (user-full-name))
  (aset the-record arch-date (current-time-string))
  the-record)                               ; return it
(setq forms-new-record-filter 'new-record-filter)

;; The format list.
(setq forms-format-list
     (list
       "====== Public Domain Software Archive ======\n\n"
       arch-shortname
       " - "                    arch-longname
       "\n\n"
       "Article: "              arch-newsgroup
       "/"                      arch-article
       " "
       '(arch-tocol 40)
       "Issue: "                arch-issue
       " "
       '(arch-rj 73 10)
       "Date: "                 arch-date
       "\n\n"
       "Submitted by: "         arch-from
       "\n"
       '(arch-tocol 79 ?-)
       "\n"
       "Keywords: "             arch-keywords
       "\n\n"
       "Parts: "                arch-parts
       "\n\n====== Remarks ======\n\n"
       arch-remarks
     ))

;; That's all, folks!
```

11 Credits

Bug fixes and other useful suggestions were supplied by Harald Hanche-Olsen (`hanche@imf.unit.no`), `cwitty@portia.stanford.edu`, Jonathan I. Kamens, Per Cederqvist (`ceder@signum.se`), Michael Lipka (`lipka@lip.hanse.de`), Andy Piper (`ajp@eng.cam.ac.uk`), Frederic Pierresteguy (`F.Pierresteguy@frcl.bull.fr`), Ignatios Souvatzis and Richard Stallman (`rms@gnu.org`).

This documentation was slightly inspired by the documentation of "rolo mode" by Paul Davis at Schlumberger Cambridge Research (`davis%scrsu1%sdr.slb.com@relay.cs.net`).

None of this would have been possible without GNU Emacs of the Free Software Foundation. Thanks, Richard!

Index

Table of Contents