(4)

VLSI Memo No. 88-487
November 1988

A USER/PROGRAMMER GUIDE TO THE *FABFORM* USER INTERFACE

Rajeev Jayavant

Abstract

Fabform is a generalized form editor which serves as an user interface for several utilities in the CAFE system. *Fabform* can be used as a means of communicating with other programs, such as the fabrication interpreter.

The basic concept of *Fabform* is simple. The user is presented with a form or template containing a number of blank fields which may then be filled in. *Fabform* displays a screen of information and allows the user to move to various fields in the form. The fields are displayed on the terminal.

This document describes the basic operation of *Fabform* for both user and programmer. Topics include how to design user interfaces, how to incorporate *Fabform* as a subroutine in user programs, and how to customize values for specific fields.

Fabform includes support for applications written in Lisp as well as C.

Microsystems Research Center Room 39-321
Massachusetts Institute of Technology
Cambridge Massachusetts 02139
Telephone (617) 253-8138

Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Agency under contract number N00014-85-K-0213.

For those who are interested in obtaining a copy of the program contact the Microsystems Research Center, MIT, Room 39-321, Cambridge, MA 02139. (617) 253-8138.

Author Information

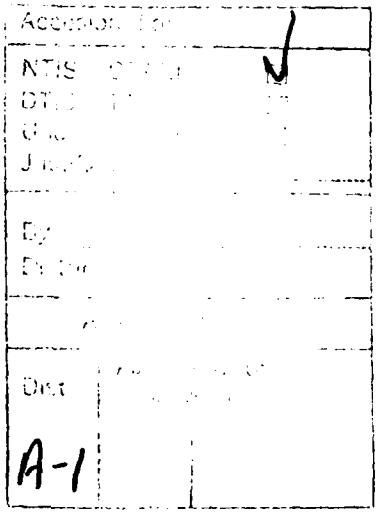
Jayavant: Department of Electrical Engineering and Computer Science, MIT, Room 36-699, Cambridge, MA 02139. (617) 253-4168.

User Guide to the *Fabform* User Interface

Rajeev Jayavant

Revised August 3, 1988

Fabform **is a generalized form editor which serves (or will serve) as the user interface for several utilities in the cafe system. This document describes the basic operation of *Fabform* from the user's standpoint. A description of using *Fabform* from a programmer's standpoint will be available in a separate document. All bug reports should be sent to "fabform@caf.mit.edu".**



Barbara Tilson (tilson@caf.mit.edu)
MIT Room 39-321
60 Vassar Street
Cambridge, Mass. 02139
Tel. (617) 253-4731

1 Introduction

Fabform is a generalized form editor which serves as the user interface for a number of utilities in the cafe system. Thus most users will never invoke *Fabform* directly, but rather will use *Fabform* as a means of communicating with some other program [1].

The basic concept of a form editor is quite simple. The user is presented with a form, or template, containing a number of blank fields which he may then fill in. *Fabform* displays a screenfull of information and allows the user to move to the various fields in the form. The fields are displayed in either reverse video or underline mode depending on the abilities of the terminal.

The value of a field may be entered or changed by the user only in accordance with a set of rules specified for the form. For example, a field in a form may only be allowed to contain floating point numbers. Another field may be constrained to contain only a member of some specified set (eg. "angstroms" or "nanometers"), and other fields may be specified as "read-only", allowing no modifications whatsoever. *Fabform* will enforce such restrictions, displaying messages if the rules are violated.

2 Basic Operation

Emacs users will notice that *Fabform* is similar to *Emacs*in terms of the screen display and the functions bound to most keys. The arrow keys are also supported for cursor movement on VT100, VT200, VT52, HEATH, and related terminals. The exact operation of *Fabform* will vary from application to application because much of the operation depends on the particular form being edited. Nevertheless, this document should provide enough information to use *Fabform* without too much trouble.

2.1 Screen Layout

Fabform will display as much of a form as will fit on a screen, allowing the user to scroll vertically through the form if necessary. The last two lines on the screen are used to display status and help information. The second last line is the status line and provides the following information:

modified flag The first few characters on the status line indicate whether the form has been altered since it was last saved. A "**" indicates that

[1] Such as the fabrication interpreter

changes were made, "--" indicates no changes, and "%%" indicates that the entire form is read-only so changes couldn't have been made. These indicators are just like those used by *Emacs*.

information area Just to the right of the modified flag is an area that can be used to display any short message chosen by the application using *Fabform*.

time The current time (in 12 hour mode) is displayed in the center of the status line and is updated once a minute.

relative position The right half of the line contains a small indicator of the relative position of the displayed screen to the top of the form. The position is specified as in *Emacs*and may be "Top", "Bot", "All", or a percentage representing the position of the first line displayed on the screen relative to the first line of the form.

The last line on the screen normally displays some help information relating to the field that is currently chosen by the cursor. If an error occurs, the last line of the screen is used to display the error message until the another key is pressed.

2.2 Command Summary

More detailed descriptions of some of the commands can be found in later sections of this document. The basic commands are:

ctrl-A if the field can only contain a set of restricted choices, advance to the next valid choice.

ctrl-B move back one field

ctrl-F move forward one field

ctrl-G abort command (really only useful in extended commands)

ctrl-L
- Move currently selected field to the center of the screen.
- If the field is already at the center, refresh the display.

ctrl-M or **RETURN**
- Enter default value if the field has a default.
- Enter current time and date if the field is a blank date field
- Otherwise just advance to next (preferably editable) field.

ctrl-N move down to field on next line

ctrl-P move up to field on previous line

ctrl-Q if the field can only contain a set of restricted choices, reverse to the previous valid choice

ctrl-R scroll screen down one line

ctrl-S scroll screen up one line

ctrl-U enter a repeat count for any cursor movement command

ctrl-V scroll screen forwards by 3/4 of the screen height

ctrl-W delete field

ctrl-X introduce extended commands

ctrl-Y restore field to the value it had before it was visited

ctrl-Z suspend the editor and return to the shell. The editor can be restarted using the "fg" command

ESC introduce more extended commands

DELETE Delete backwards one character.

ctrl-X extended commands

ctrl-G abort command

ctrl-X ctrl-C exit editor. If the form has been modified, the user is asked whether the changes should be saved. If all "required" fields have been filled in, the user is asked whether he has completed all work on the form. If all work is not complete, the editing can be resumed from the point the editor is exited.

Some applications use *Fabform* in *impatient mode*, in which case the user will not be asked whether he wants to exit or save changes. All changes will automatically be saved and *Fabform* will exit.

ctrl-X ctrl-S save the state of the edit session

ctrl-X ctrl-T like ctrl-X ctrl-C but allows the user to declare that he is through editing a form even if some normally required fields are left blank. An edit session terminated via ctrl-X ctrl-T normally will not be continuable whereas one terminated by ctrl-X ctrl-C will be.

ctrl-X ctrl-W write a printable description of the current state of the form to a file. *Fabform* will prompt for the filename to use. Full pathnames must be specified if the file is to reside in any directory other than the current working directory or one of its decendants.[2] The file can later be printed using **lpr** or displayed using **more** or **ul**. ctrl-X ctrl-W is only useful for producing some printable output from *Fabform* – ctrl-X ctrl-S must still be used to save the state of the edit session.

ctrl-X ctrl-Z suspend the editor and return to the shell. The editor can be restarted using the "fg" command.

ctrl-X ? display help information

ESC extended commands

ctrl-G abort command

ESC < move to beginning of form

ESC > move to end of form

ESC V scroll screen backwards by 3/4 of the screen height

ESC X execute a shell command. The user is prompted for the command to be executed.

ESC DELETE delete backwards until a whitespace is found.

ESC ? display help information

[2] ie. Filenames should not begin with a ~ unless the actual name of the file (as displayed by ls) does.

3 Cursor Movement

The cursor can be moved from field to field using either the arrow keys or ctrl-B, ctrl-F, ctrl-N, and ctrl-P. The ctrl-N and ctrl-P commands attempt to place the cursor on a field directly above or below the current field. If a suitable field cannot be found, the next-closest field is chosen.

The ctrl-V, ESC V, ctrl-R, and ctrl-S commands attempt to keep the cursor on the current field while scrolling through the form. If the current field is moved off the screen, an attempt is made to move to a field on the new screen closest to the previously chosen field.

The ESC < and ESC > commands move to the first and last fields in the form, respectively.

If there is no visitable field on a screen, the cursor will move to the lower right corner of the display, and all editing operations will be disabled until a field is brought back on the screen using one of the cursor movement commands or the ctrl-L command.

The RETURN key can also be used to advance from field to field. The direction of the advance is dependent upon the particular application – some applications prefer advancing towards the right to the next editable field while others advance downwards to the next visitable field. Care must be taken in using the RETURN key to advance the cursor since it will also fill in default values of fields (refer to upcoming sections on default and date fields).

4 Editing Field Entries

There are only four commands available for editing the value entered in a field. DELETE deletes the last character in the field while ESC DELETE deletes characters backwards from the end of the field until a space is found. The entire field can be deleted using ctrl-W, and ctrl-Y can be used to return a field to the value it had before the cursor was moved onto it.

5 Validation of Field Entries

Fabform recognizes a number of field types and restricts a user's input to conform to the requirements of the field. Thus, the actions *Fabform* takes while the user is entering a value into a field depends upon the characteristics of the field.

5.1 Unrestricted Fields

Unrestricted fields may contain any printable character (and spaces). Of course, the length of the field is also a limiting factor.

5.2 Integer Fields

An integer field can be composed of any string of digits with an optional preceding "-". Attempting to enter anything else will result in an error message and the erroneous input will be disregarded.

5.3 Floating Point Fields

A floating point field can be composed of digits, ".", "+", "-", and "e" or "E". If any other character is typed, an error message will be displayed. An error will also occur if the entry is not a properly formatted floating point number (optionally in scientific notation) and an attempt is made to either move to another field or save the field. The entered value is also checked for overflow.

5.4 Lisp Expression Fields

Basically any valid printable representation of a lisp object (symbol, number, list, string, etc.) may be entered. *Fabform* will flash the matching open parenthesis when a close parenthesis is entered.

5.5 Oneof Fields

Oneof fields can only contain one out of a specified set of choices. A user's input into a oneof field is checked against all possible choices as the entry is made. If *Fabform* detects that the user is not entering a valid choice, an error message is displayed. If, on the other hand, the user enters enough characters to uniquely specify a valid choice, the remainder of the choice is automatically entered. *Fabform* will also attempt to complete partial matches of the user's input to the available choices, thereby minimizing the amount of typing required. Case conversion is performed on the user's input, if necessary, to match the valid choices.

If the user does not know what the possible choices are, or if he just wants to advance through them one by one, the ctrl-A and ctrl-Q keys may be used. If the choice has been partially, but not uniquely, specified,

pressing ctrl-A or ctrl-Q will display the first choice that matches the partial specification.

5.6 Default Values

All of the field types described above may have an associated default value. The help line at the bottom of the display will contain "(default available)" whenever a field with a default value is selected. The default value, however, is not used unless it is explicitly selected by pressing RETURN or ctrl-M when the field is selected. If a different value is entered for the field, the default is lost.

5.7 Date Fields

Date fields are similar to Default fields except that the current date/time is the default. Simply press RETURN to enter the current date into the field. Only the month and day need to be entered if the current date is not wanted. The defaults for the remaining (unspecified) parts of the date are the current year for year and zero for hours, minutes, and seconds.

5.8 Read-only Fields

Read-only fields, as their name implies, cannot be modified. When the cursor is on a read-only field, the help line at the bottom of the screen will display "(read only)". *Fabform* will display an error message if you attempt to modify a read-only field.

The read-only status of a field can change over time. A form may contain fields that change to read-only status after a save operation [3].

6 Saving Entered Field Values

The field values entered by the user are not saved until an explicit save is performed either via ctrl-X ctrl-S or during exit from *Fabform*. If the program dies for any reason (eg. due to a machine crash), any changes made since the last save will be lost. Thus it is advantageous to save often.

On the other hand, some forms contain fields that become read-only after a save. Thus extra care should be taken to ensure that the entries in such fields are correct before performing a save.

[3] eg. to prevent history from being modified

7 Exiting *Fabform*

7.1 Temporary Suspension

Fabform may be temporarily suspended using either the ctrl-Z or ctrl-X ctrl-Z commands. *Fabform* can restarted later using the "fg" command from the shell or by selecting the appropriate choice in the "Tasks" submenu in the wand.

Some applications may disable the the suspension mechanism, in which case ctrl-Z and ctrl-X ctrl-Z will only generate a warning message. In the absence of the suspension mechanism, the only way to exit *Fabform* is via one of the quit commands.

The most common reason for using a suspension mechanism is to temporarily stop a process, execute some other commands in the shell, and then resume the original process. If the suspension mechanism is disabled, shell commands can still be run using the ESC X command.

7.2 Permanent Exit

There are two commands for permanently exiting *Fabform*: ctrl-X ctrl-C and ctrl-X ctrl-T. In many applications, there is no difference between the two commands, with ctrl-X ctrl-T defaulting to ctrl-X ctrl-C. In most cases, ctrl-X ctrl-C should be used to exit *Fabform* – ctrl X ctrl-T is normally used only when the user has finished all editing [4] of a form but wishes to leave some fields blank. The properties of the two commands can be summarized as follows:

ctrl-X ctrl-C This can be thought of as a long term suspension. If any changes have been made, the user is asked if the changes should be saved before exiting (allowing a quick way to abandon any changes made to the form). *Fabform* terminates after confirming the user's intentions [5] , but indicates to its associated application that editing of the current form is to be resumed at some later time. If, however, all fields in the form have been filled in, ctrl-X ctrl-C defaults to ctrl-X ctrl-T.

[4] ie. he never wants to edit that particular form again

[5] except when an application uses *Fabform* in *impatient mode*. All changes will automatically be saved and *Fabform* will exit without confirming intentions.

ctrl-X ctrl-T *Fabform* saves all changes and exits after confirming the user's intentions.[6] The associated application is *told* that the user has completed all editing on the current form and does not wish to return to it at a later time. If some fields have not been filled in, the user is warned of their existence and asked for a confirmation of the command.

[6] except when *impatient mode* is in effect

Programmer's Guide to the *Fabform* User Interface

Rajeev Jayavant

Revised August 3, 1988

Fabform is a generalized form editor which serves as the user interface for several utilities in the cafe system. This document describes the basic operation of *Fabform* from the programmer's standpoint, providing the information necessary to use *Fabform* as the user interface for a given application. The User's Guide to *Fabform* should be consulted for further information. Questions and bug reports should be sent to "fabform@caf.mit.edu".

Barbara Tilson (tilson@caf.mit.edu)
MIT Room 39-321
60 Vassar Street
Cambridge, Mass. 02139
Tel. (617) 253-4731

1 Introduction

Fabform is a generalized form editor designed to be used as the user interface for a variety of programs. It is quite flexible in terms of the types of forms that can be handled, though the attempt to keep *Fabform* as generalized as possible imposes a few limitations. Terminal independence is an important feature of *Fabform*, though performance degrades rapidly when running on terminals with fewer capabilities than a vt52. This document describes the capabilities and deficiencies of the current implementation of *Fabform*.

Software updates and bug fixes are made periodically. Every attempt will be made to keep new releases of *Fabform* completely compatible with applications designed around older versions. Please send a bug report if you suspect that a problem is introduced in a new release that makes it incompatible with previous versions. If you feel that your application (and hopefully others) could benefit significantly by an addition of features to *Fabform*, please send mail to "fabform@caf.mit.edu" and describe your situation. Perhaps the feature can be added if the modification is feasible.

2 Interaction with *Fabform*

Fabform is designed to run as a separate process and communicates with the associated application program via two files: the template file and the parameter file. The template file describes the layout of the form and the types of data that can be entered in the various fields. The parameter file specifies the contents of the various fields defined in the template file. *Fabform* takes a template and/or parameter file as input and produces a parameter file as output. Both input and output parameter files may be read from/written to pipes if desired. The `examples` directory under the *Fabform* source code directory contains some sample application programs that interact with *Fabform*.

Once the application program invokes *Fabform*, it can no longer communicate with *Fabform*. All interaction between the two processes is done only through the files described above, and the input template and parameter files are read only when *Fabform* is invoked. The output parameter file, however, can be read at any time, so the application can keep track of changes

to the output file as they happen[1]. If the output parameter file is actually an output pipe, the process reading the information sent down the pipe can monitor data as it is saved by the user.

The *format* of the output parameter file is identical to that of the input parameter file (described later). The actual ordering of information within the output parameter file can vary greatly from that in the input parameter file, thus programs reading the parameter files should not make any assumptions about the ordering of information. The only ordering that is guaranteed in the output parameter file, in addition to the ordering imposed by the restrictions on parameter files, is that multiple instances of an operation appear in the same order as they appear in the form being edited. If the output parameter file is actually a pipe, the EOF character (control-D or '\004') is used to indicate the end of a complete set of data[2].

Whitespace is added to the output parameter file in an attempt to make it more readable (presumably for debugging purposes). No assumptions should be made about the existence of whitespace since the specifications of the parameter file format state that whitespace should be ignored.

The exit status of *Fabform* determines whether the user has completed all editing of the form.[3] Exit status 0 indicates that all editing is complete, status 1 indicates the user wishes to return to the form at some later time, and a negative status indicates a fatal error occurred[4].

The template files are intended to be static in the sense that under most circumstances they would be created once and used repeatedly. Parameter files, on the other hand, are normally created dynamically as they are needed

[1]The output parameter file is first written as soon as *Fabform* finishes reading the input parameter file and is updated whenever the user executes a *save* command

[2]The output parameter file is updated whenever the user executes a *save* command. If the output is to a pipe, there is no way to *rewind* the file to the beginning, thus the EOF character is used to indicate the end of a block of data that would normally correspond to a complete output parameter file.

[3]ie. Does the user want to resume editing the current at some later time or has he/she made all the modifications that are ever to be made to the form? A form is said to be complete when the user has entered values in all fields that are required to be filled in, or if the user claims that he has completed filling in the entire form.

[4]Fatal errors can be caused by errors in the template or parameter files, by failure to open the specified files, refusal of the system to grant a memory allocation request, or a fatal error in *Fabform* itself.

and deleted when they are no longer needed.

3 Template File Format

The template file format resembles LaTeXfiles in that all command directives begin with a "\" and any amount of blankspace[5] is treated as a single space. A \ may be included in the form by specifying \\ in the template file.

3.1 Positioning Commands

\hspace[#chars] insert horizontal blankspace of #chars

\hpos[column#] move to horizontal position given by column#. First column is 1

\nl begin new line

\vspace[#lines] insert #lines blank lines. \vspace[0] is equivalent to \nl

Any blankspace in the file before or after \hpos or \hspace is ignored. Blankspace is also ignored if it occurs at the beginning of a line.

3.2 Field Definition Commands

Field definitions consist of a field width, a field name[6], and some help information describing the purpose of the field. Some fields implicitly define the width while others do not have the help information because users are not allowed to move the cursor onto those fields. The help information is normally displayed on the last line of the screen when the user places the cursor on a field. The restriction on field width is determined at compile time, though the typical maximim field width is over 2000 characters. Thus there is no problem having fields that span line breaks.

[5] Blankspace is defined as any combination of spaces, tabs, and newlines.

[6] Field names will be also be referred to as *tags* or *parameters* in later sections of this document.

The field declaration describes the type of data that may be entered into a field. The parameter file may place further restrictions on the type of data that may be entered.

The field definition commands are:

\global[width][name] insert global parameter for specified width. A global parameter is one that can be accessed from within any operation block (see next section) but cannot be entered by the user. If the value of a global is not specified in the parameter file, the field is left blank. There are a few globals that are defined by *Fabform* if they are not specified in the parameter file.

user the login name of the user running *Fabform*

now the time at which *Fabform* was invoked. The time is in the format "mm/dd/yy hh:mm:ss"

editor-name some default string that is printed on the status line

help-file the file containing a summary of *Fabform* commands

\integer[width][name][help] allow only the digits 0 through 9 with an optional preceding minus sign

\float[width][name][help] allow only valid floating point numbers, including scientific notation

\string[width][name][help] allow any arbitrary string of printable characters

\comment[width][name][help] just like \string except that the field does not become read-only after a save. This field type is equivalent to \string*- or \string-* (see below) and exists only for backward compatibility.

\readonly[width][name][help] just like \string but always read-only (but visitable, unlike \hidden)

\lispexp[width][name][help] Allow only valid "lisp" expressions. Tests for matching double-quotes and parentheses. The matching open parenthesis is flashed whenever a close parenthesis is entered.

\hidden[width][name] If the value is not specified in the parameter file, the field is left blank. The user cannot move the cursor onto a hidden field. Very similar to a global except that the name lives in the local namespace of the operation (see next section).

\date[name][help] user can enter the current time into the field by pressing RETURN. The date is in the format "mm/dd/yy hh:mm:ss". If the current time is not desired, the user may enter any valid date and time. Only month and day need to be specified with the year defaulting to the current year and hours, minutes, and seconds defaulting to zero.

\day[name][help] Just like \date except that the field does not include the time of day. The day is in the format "mm:dd:yy". The strange naming of these two field types is the result of preserving backward compatibility.

\global, \hidden, and \readonly fields are always read-only (and only \readonly is visitable) whereas \comment fields are always writable. All other field types are write-once; the fields are writable until the user enters a value and *saves* the state of the form. Appending a "*" to any field type forces the field to remain writeable after a save. Appending a "-" to any field type signifies that filling in the field is optional, thus the field is considered to be filled when *Fabform* determines whether the user has completed filling in the form.

3.3 Operation Block Delimiters

All tags other than the globals live in the local namespace of the operation block they are defined in. Thus it is possible to reuse tag names as long as they occur in different operation blocks. There may be multiple instances of the same operation block, but operation blocks may not be nested. Names of tags, globals, and operation blocks (described below) are case-insensitive.

An operation block must be started before the first non-global field can be defined, and an operation block must end before the next one can begin.

The block delimiting commands are:

\begin[operation_name] defines the start of a new operation block.

\end[operation_name] defines the end of an operation block. The operation_name specified must match the name specified in the \begin command.

4 Parameter File Format

The parameter files use a lisp-like syntax to delimit different operation instances that may exist within a single parameter file. The structure of the files is quite flexible. Whitespace (spaces, tabs, and newlines) is ignored except when it occurs within double-quotes ("). Since double-quotes are used to delimit strings, they must be preceded by a \ if they are used within a string. Similarly, a \ is represented as \\ within a quoted string. Information within a parameter file is case-insensitive (except for quoted strings) and will be represented in lower-case in the output parameter file. Additional restrictions are described at the end of this section.

The following three strings are keywords and should be avoided as tag names to prevent any possible confusion. None of these directives may be nested within each other.

template specifies a template file to use to extend the current form

operation defines a block of parameters for an operation instance. Details on syntax follow. Parameters declared/defined within a particular operation instance remain local to that instance, allowing multiple instances of the same operation.

define-oneof-class define a class of objects that can later be specified using the defined class name (see **oneof-class**).

The following directives may be specified only within parameter definitions within an *operation* directive. These directives are not guaranteed to work if they are nested, but they may be used in conjunction with each other as described later in this section.

default defines a default value for a parameter that does not have a value assigned to it. The default value does not become the value of the parameter until the user visits the field and presses RETURN.

initial-value defines an initial value for a parameter and allows the user to modify it. This differs from simply giving a value to a field in that fields that become read-only may be given an initial value that can be modified.

private assigns a value but does not allow the user to tell whether the value is really a parameter or something hardwired into the template. Good for hiding things from users who are better off not knowing the truth.

oneof restrict the parameter to take on one of the specified values

oneof-class like **oneof** but specifies the name of a class defined via **define-oneof-class** rather than a list of choices

exec-function function to execute when the value of the field is changed. Refer to the Addendum to the Programmer's Guide to Fabform for more details

In addition to the above directives, *Fabform* supports a class of values called globals that may be accessed from any part of a template file. Globals are defined at the start of a parameter file using the syntax:

```
(global-parameter "value")
```

Fabform also provides default values for four globals:

user the login id of the user

now the time at which *Fabform* began execution

editor-name the message displayed in the status line. default is set at compile time at the whim of the programmer.

help-file the name of the file to display when the user asks for help. Normally this is a file containing a summary of *Fabform* commands.

A parameter file may define globals that do not exist anywhere in a template (allowing information to be transferred from the input parameter file to the output parameter file), but any local field definitions (ie. within operations) must correspond to fields defined in a template file.

The *template* directive allows many separate template files to be appended together to create a single form. Each *template* directive causes a separate template file to be appended in the creation of the form. The syntax is:

```
(template "template-file-spec")
```

The *operation* directive takes the form

```
(operation operation-name
        <parameter value assignments>
        ...
)
```

The *define-oneof-class* directive is useful if many fields are to be restricted to the same set of values. Rather than specifying the restriction list explicitly within each parameter description via a oneof declaration (see below), the restriction list can be defined once using the *define-oneof-class* using the form

```
(define-oneof-class class-name choice_1 choice_2 ...)
```

The defined class can then be referenced via the *oneof-class* declaration within a parameter definition. Class names defined using *define-oneof-class* live in their own namespace, but this namespace is common to all operation instances. Thus class definitions are global to all procedures and a class may not be redefined. A class name may, however, be the same as that of a global or a tag within an operation.

Parameter values are assigned using one of the following forms:

```
        (parameter-name value)
        (parameter-name (default value))
        (parameter-name (initial-value value))
        (parameter-name (private value))
        (parameter-name (oneof choice_1 choice_2 ... choice_n))
(parameter-name (oneof-class class-name))
```

where value may be any arbitrary string. Whitespace will be deleted unless value is enclosed in double quotes. The double quote character therefore may not be included as part of any value. All values will be truncated to the field width specified in the template file, if required. The *oneof* (or *oneof-class* and *default* directives may be used in conjunction, e.g.

```
(parameter-name (default-value)
                (oneof choice_1 choice_2 ... choice_n))
```

In the case of fields that do not become read-only after a save, it is even sensible to specify a value in addition to a *oneof* declaration, eg.

```
(parameter-name value
                (oneof choice_1 choice_2 ... choice_n))
```

Parameter names may also be specified hierarchically, eg.

```
(measurements
    (right
        (value 2378)
        (units "A"))
   (left
        (value 2412)
        (units "A")))
```

Internally the parameter names are treated as "measurement.right.value", "measurement.right.units", "measurement.left.value", and "measurement.left.units". These expanded names are the ones that must be used within the template file.

A parameter file consists of any number of *global definitions* followed by any number of *operation*, *template*, and *define-opset-class* directives. The only restrictions are that all *global definitions* must appear before any *template* directives, the *template* directive corresponding to an *operation* directive must occur before the *operation* directive, and a *define-opset-class* directive must occur before the class-name is referenced in a *oneof-class* declaration. The number of operation and template directives is largely determined by the capability of the VM system and the patience of the user for going through long forms.

The format of the parameter can be parsed easily by a Lisp application, but C applications may find the job more difficult. To ease the burden for C programmers, a collection of routines for parsing parameter files is provided in the `utilities` directory under the *Fabform* source directory, along with the accompanying documentation.

5 Invoking *Fabform*

The *Fabform* command line syntax is as follows:

```
/usr/cafe/lib/fabform [-p input_parameter_file]
                      [-o output_parameter_file]
                      [-d template_directory] [-t template_file]
                      [-f output_form_file] [-r] [-v] [-q] [-n]
      [-s field_name:op_name:op_instance]
                      [-H help-file] [-N editor-name]
                      [-z] [-i] [-e] [-D]
```

Any or all of the flags may be specified. Either an input parameter file or a template file must be specified in all cases. The command line flags are interpreted as follows:

-p Use the next argument as the name of the input parameter file. If a template file is also specified with the -t flag, the template file will be read first.

-o Use the following argument as the output parameter file. The output file may be the same as the input file without any chance of data loss. The output file is written whenever the user performs a save operation and is guaranteed valid unless there is a shortage of disk space or the system crashes during a write.

If the output filename begins with a "|", pipe the output to that process instead of writing it to a file. For example,

```
fabform -p infile -o "|foo arg1 arg2"
```

will pipe the information that would be sent to the output parameter file into the process "foo arg1 arg2". An EOF character (control-D, or '\004') will be sent down the pipe to indicate the end of a save operation (since the user may repeatedly save his data).

-d The next argument will be used as the directory in which to look for template files. A trailing / will be added if necessary.

-t Use the following argument as the name of the template file to read. If an parameter file is also specified, the template file will be read first. The -t option is useful for generating skeleton parameter files from a template file (eg. "fabform -t templatefile -o outputparamfile -f /dev/null") or for using different templates with a static parameter file.

-f Use the next argument as the filename to store a printable representation of the form into. The file can then be printed using *lpr* or displayed on the screen using *ul* or *more*. *Fabform* does not start up interactively and simply exits after creating the form file.

-n The default movement upon pressing RETURN is changed to vertical instead of horizontal

-q Do not attempt to report whether all editing is complete. Normally *Fabform* will attempt to determine whether all editing is complete on a given form and return an appropriate exit status, sometimes by querying the user. In situations in which the notion of completeness does not apply, the -q option should be used to prevent the user from being unnecessarily harassed. If -q is specified, ctrl-X ctrl-T behaves like ctrl-X ctrl-C, and the exit status of *Fabform* is meaningless if it is non-negative.

-r Read-only mode. Allow the user to browse through the form without changing anything. No output parameter file is created even if -o is specified.

-H specify the file to print as the help file. Can be overridden by a specification in the parameter file.

-N specify the text to use for the "editor name". Can be overridden in the parameter file.

-z Disable suspension via ctrl-Z or ctrl-X ctrl-Z. Useful if you don't want the user to suspend the application.

-D Display messages for non-fatal errors in parameter or template files. If any such errors are detected, *Fabform* terminates after reading the entire parameter or template file. If the -D flag is not specified, non-fatal errors are not reported and operation of continues as usual. The non-fatal errors are:

- Definition of parameters not used in the corresponding template file. The definition is ignored.

- Specifying an undefined *class* in a **oneof-class** declaration. The **oneof-class** declaration is ignored, and no additional restrictions are put on the field.
- Multiply defining a *class* in **define-oneof-class** declarations. Only the first definition is stored.
- A reverse \hpos while reading a template file

-i Use *impatient mode*. Do not verify user's intentions for ctrl-X ctrl-C, ctrl-X ctrl-S, or ctrl-X ctrl-T. A save is automatically performed before an unconditional exit.

-e Initially position cursor on last editable field in the form instead of the first.

-s Use the next argument to determine where to position the cursor at the start of the edit session. The cursor will start out on the field named `field_name` in the `op_instance`th instance of the operation named `op_name`.

input pipe The input parameter file can be read from an input pipe rather than a file. The parameter file must be *piped into Fabform* using a mechanism similar to the **popen()** function or the piping mechanism of a shell. The **-p** option overrides use of an input pipe to read the input parameter file.

output pipe The output parameter file can be written to an output pipe rather than a file. The output of *Fabform* can be *piped into* another process using a mechanism similar to the **popen()** function or the piping mechanism of a shell. *Fabform* can use this mechanism to pipe the output parameter file to an existing process (eg. its parent). The **-o** option will override the use of an output pipe[7]

[7] An output pipe created using the -o option first creates a new process into which the output parameter file will be piped. Using an output pipe without the -o option allows communication with an existing process.

Addendum to Programmer's Guide to the *Fabform* User Interface

Rajeev Jayavant

Revised July 1, 1988

Fabform is a generalized form editor which serves as the user interface for several utilities in the cafe system. This document describes how *Fabform* may be incorporated as a subroutine in a user program. *Fabform* now includes support for applications written in Lisp as well as in C.

The Programmer's Guide to *Fabform* contains additional information that is required in order to use *Fabform* within an application. The User's Guide to *Fabform* should be consulted for further information. Questions and bug reports should be sent to "fabform@caf.mit.edu".

Barbara Tilson (tilson@caf.mit.edu)
MIT Room 39-321
60 Vassar Street
Cambridge, Mass. 02139
Tel. (617) 253-4731

1 Introduction

One major disadvantage of running *Fabform* as a separate process in the minimal interaction between the application and *Fabform* while the user is editing the form. By incorporating *Fabform* into the application itself, it becomes possible to associate a function [1] with each field. The function will be called whenever the value of that particular field changes [2]

The called function may modify the behavior of *Fabform* upon return in any of the following ways:

- change the value of the current field
- display a message in the status line
- make the current field writable again (if it was made read-only by a save just prior to calling the function)
- save the output parameter file
- exit *Fabform*

In addition, the function may perform any actions it wants to, including writing to the user's terminal or modifying the signals. To avoid confusing *Fabform* upon return, mechanisms are provided to instruct *Fabform* to reset its signals and refresh the screen.

A final bonus is that *Fabform* may be called recursively with minimal overhead since there is no need to start a new process. Of course, if very large forms are used, there is a chance of running out of memory space.

The only real disadvantage to including *Fabform* as a subroutine in an application is that the application must be relinked to incorporate the latest bug fixes and enhancements made to *Fabform*. [3] When *Fabform* is run as a separate process, the most recent version is automatically available.

[1] The functions may be written in C or Lisp, and all of the functions do *not* have to be written in the same language.

[2] The function declaration specifies whether to call the function when a new value is "entered" into the field or to call it as individual characters are entered into the field.

[3] A Lisp application may choose to load the *Fabform* package at runtime, thereby automatically obtaining all updates.

2 The *Fabform* Interface

There are two routines which may be executed to start a *Fabform* edit session, `fabform()` and `fabforml()`. The routines differ only in the format in which the arguments are specified. Lisp applications must use the macro `fabform:exec-fabform` which takes very similar arguments.

```
int fabform(function_defs,args)
    struct fabform_functions *function_defs;
    char *args[];

int fabforml(function_defs,arg1,arg2,...,argn,(char *)0)
    struct fabform_functions *function_defs;
    char *arg1,*arg2,...,*argn;

(fabform:exec-fabform function-definition-list arg1 ... argn)
```

2.1 Function Definitions

The `function_defs` argument specifies a list of functions, names, and instructions for running those functions. The `function-definition-list` in the Lisp macro performs the same role. The actual assignment of a function to a field in the form is done via the input parameter file using the `exec-function` construct. For example, to attach the function named "twiddle" to a field named "knob", the parameter file entry would look like:

```
(knob (exec-function "twiddle"))
```

Of course, other assignments can simultaneously be made to "knob" (e.g. oneof, oneof-class, default, private, or even a value). The name given in an `exec-function` construct must match a name in the `function_defs` list passed to *Fabform*, otherwise no function assignment will be made to the field.

The format of the `function_defs` entries is as follows:

```
struct fabform_functions {
   struct fabform_retstat (*function)();
   char *name;
   int action;
 }
```

The fields in the structure have the following meanings:

function a pointer to the function to be executed. The calling interface of these functions will be discussed later in this document.

name the name that this entry will be referenced by in an `exec-function` construct in a parameter file

action the actions that will be taken *before* the function is called. The function may specify additional actions to be taken upon returning (discussed later). The actions field should be constructed by ORing (use |) any of the following values together.

FABFORM_SET_SIGNALS resets SIGINT, SIGQUIT, SIGTSTP, SIGBUS, SIGSEGV, and SIGALRM to their original handlers. The signals will automatically be restored to *Fabform's* preferred settings upon return. Any function that wants to take control for an extended period (e.g. to interact directly with the user or start up another process) should use this feature. See section on Signal Handling for more details.

FABFORM_SAVE save the state of the form into the output parameter file before calling the function. Allows the function to look at the values of other fields in the form. Be careful with this one since some fields can become read-only after a save.

FABFORM_EXIT exit *Fabform before* calling the function. This can be useful if the function always exits *Fabform* and has the advantage that memory allocated by *Fabform* is released *before* the function is called.

FABFORM_NO_REFRESH do not save the tty state and keep the cursor in the current position. This value should only be included if the function is not going to affect the screen in any way whatsoever. If FABFORM_NO_REFRESH is missing, the

tty state is saved and the cursor is moved to the lower left corner. A screen refresh is performed on return only if the return status requests one.

FABFORM_CLEAR Clear the screen before calling the function. A refresh will be performed upon return even if one is not requested.

FABFORM_WRITABLE keep this field writable even if it is made readonly by a save

FABFORM_REEDIT keep cursor on present field after return

FABFORM_ANY_KEY call the function any time the user presses a key that leaves a valid value in the field[4]. Normally the function is called only when the user "enters" a value in a field (e.g. by pressing RETURN or moving to another field).

FABFORM_EXEC execute function. This one will automatically be set if it is missing. Kind of useful to have something to assign, though, if none of the other options are desired.

The final entry in the `function_defs` list must have the name field set to `(char *)0`. If no function definitions are to be made, `function_defs` may be specified as `(struct fabform_functions *)0`.

The Lisp `function-definition-list` is used to express the same information as `function_defs` is in C. The `function-definition-list` is a list of function definitions of the form:

```
(function-pointer function-name action)
```

The fields of the function definition (which is also a list) are very similar to their C counterparts.

function-pointer A Lisp integer that may be passed to C as a pointer to an executable C function. A suitable pointer to a Lisp function is obtained by using the macros `fabform:make-fabform-function` or `fabform:defun-fabform-function` to define a Lisp function (discussed in the next section).

[4]The function is not called when the *new* value is obtained by deleting characters. This is currently considered a *feature* and may not be present in future implementations.

function-name The name the function will be referenced by in the `exec-function` entry of a parameter file.

action A list composed of the following keywords whose actions correspond to those of their C counterparts.

- :clear
- :set–signals
- :save
- :exit
- :no–refresh
- :writable
- :reedit
- :any–key
- :exec
- :no–exec

The action list may be nil.

If the first entry in the `function_defs` list is named "fabform_init", the function declared in that entry will be called before any editing is begun but after all field definitions and values have been read from the specified files. The "fabform_init" function is like any other function associated with a field except that all the parameters it is called with will be passed in as NULL or 0 when it is called as an initialization routine.

2.2 *Fabform* Options

The arguments specified in `args` or `arg1` through `argn` are equivalent to the command line options for *Fabform*. If the `fabform()` routine is used, the last entry in `args` should be `(char *)0` to indicate the end of the list.

`fabform1()` provides a simpler interface to use if the number of arguments is known at compile time. There is a limit of about 100 arguments for this interface while the number of entries in `args[]` for `fabform()` is unlimited.

All input to *Fabform* is still in the form of parameter files and template files. Applications making full use of functions associated with fields, however, may find it unnecessary to use an output parameter file. Such applications should simply not specify an output file and save everyone from doing unnecessary work.

2.3 Return Value

The value returned by `fabform()` and `fabforml()` is identical the exit status returned by a *Fabform* process, except if a function requests an exit. Briefly, the exit status can be:

<0 fatal error. an error message describing the problem should be generated

0 user exited form and claims that all entries are filled

1 user exited form and wishes to re-edit at a later time

>1 a function associated with a field requested the exit. The actual return value is meaningless to an application.

3 The Function Interface

The functions associated with fields must be of the form:

```
struct fabform_retstat
     function_to_exec(field_value,field_name,operation_name,
                      operation_instance,output_file_name)
            char *field_value,*field_name,*operation_name;
            int  operation_instance;
            char *output_file_name;
```

where the operands have the following meanings:

field_value the contents of the field

field_name the name of the field whose contents were changed. Allows the same function to be associated with many different fields. The field_name will always be in lower case.

operation_name the name of the operation in which the field is defined [5] The operation_name will always be in lower case.

operation_instance indicates which instance of the operation the field is defined in [6]. The first instance of an operation is numbered 1.

output_file_name the name of the output parameter file in case the function is interested in the state of globals, etc. Much more useful if the FABFORM_SAVE attribute is set in the `function_defs` entry for the function. The values of other fields and their restriction lists can be examined and modified via a set of functions.

The return value of the function specifies the actions that *Fabform* should take. The `fabform_retstat` structure is defined as:

```
struct fabform_retstat {
  char *field_value;
  char *message_string;
  int action;
}
```

All of the fields must be assigned using the following guidelines.

field_value The contents of the field can be changed by specifying a pointer to a character string in the `field_value` entry of the return value. The specified string should be in a static location, not in the stack frame of the called function since this stack frame will be destroyed upon return. To leave the contents of the field unchanged, assign the `field_value` entry to `(char *)0`.

message_string *Fabform* will display the string that the `message_string` entry points to. Once again, the string should be in a static loca-

[5] It is possible to use the same field name in different operations (or \begin[]/\end[] blocks).

[6] An operation name does not uniquely identify an operation instance; an operation of a given name can occur any number of times within a form.

tion and not in the stack frame of the called function. If the `message_string` entry is assigned to `(char *)0`, no special message will be displayed.

action Specifies what actions *Fabform* should take upon the function's return. If no special action is to be taken, the `action` entry should be set to FABFORM_NO_EXEC. Otherwise it should be set to the logical OR of any of the following values.

FABFORM_SET_SIGNALS restores signals to a state that *Fabform* prefers them to be in. This option should be used anytime a function changes signal handlers, though it is unnecessary if FABFORM_SET_SIGNALS was specified in the `function_defs` entry for the function. When in doubt, specify FABFORM_SET_SIGNALS.

FABFORM_SAVE save the state of the form into the output parameter file. Can be handy if used in conjunction with FABFORM_EXIT. Be careful with this one since some fields can become read-only after a save.

FABFORM_EXIT exit *Fabform*. Allows quick exits without having to enter ctrl-X ctrl-C all the time.

FABFORM_NO_REFRESH do not refresh the screen upon return. Normally the screen is refreshed to ensure integrity of the display, but any function that can guarantee no output to the terminal may set this value to avoid unnecessary refreshes.

FABFORM_WRITABLE keep this field writable even if it is made readonly by a save

FABFORM_REEDIT keep cursor on present field after return

3.1 The Lisp Function Interface

Lisp functions associated with fields must be of the form

```
(defun function-to-exec (field-value field-name
operation-name operation-instance
output-file-name)
        (...)
```

```
...
(fabform:return-values new-field-value
                       message-string action))
```

The parameters are identical to their C counterparts. Since there is a problem in returning values directly from Lisp to C, the Lisp function must call `fabform:return-values` to return values to *Fabform*.[7] The `new-field-value` and/or `message-string` arguments should be `nil` if the field value and/or message string are not to be modified upon return. The `action` field should be a (possibly null) list of keywords from the following set (:set–signals, :save, :exit, :no–refresh, :writable, :reedit, :any–key).

The Lisp function can be made callable by Fabform by using value returned by

```
(fabform:make-fabform-function function-to-call)
```

in the function definition list passed to `fabform:exec-fabform`. Since `fabform:make-fabform-function` creates a new function and registers it in a table of C-callable functions, it should not be called repeatedly for the same Lisp function to avoid overflowing the table.

Alternately, the macro `defun-fabform-function` can be used. It uses the same format as `defun` but returns a value that may be used in a function definition list passed to `fabform:exec-fabform-function`.

4 Signal Handling

Fabform uses its own signal handlers for SIGINT, SIGQUIT, SIGTSTP, SIGALRM, SIGBUS, and SIGSEGV. The SIGTSTP, SIGBUS, and SIGSEGV handlers invoke the original handlers after doing their thing.

Whenever *Fabform* sets these signal handlers, it first checks to see whether the presently assigned handler is different from what it wants to use. If the old handler is different, it is saved for later use when *Fabform* resets signals.

[7] The *real* return value of the Lisp function should be the value returned by the call to `fabform:return-values`.

Functions associated with fields may modify signal handlers under two conditions:

1. The FABFORM_SET_SIGNALS bit is set in the `action` field of the returned value. This will tell *Fabform* to reinstall its own signal handlers.
2. The function must be aware that any signal handler that it changes becomes the new *saved* handler which will be reinstalled whenever *Fabform* resets the signal handlers.

Fabform will reset signal handlers (ie. reinstall the saved handlers) upon a normal exit. Any functions which do a recursive call to *Fabform* should be sure to set the FABFORM_SET_SIGNALS bit in the `action` field of the return value[8]. Upon abnormal exit from *Fabform*, everything is restored to the state that existed when that invocation of *Fabform* was begun.

5 Utility Routines

The following routines may be called from functions associated with fields. They allow simple methods by which data or routines internal to *Fabform* may be accessed while retaining the safety of a level of abstraction.

```
char fb_display_message(message)
        char *message;

(fabform:display-message message)
```

Display the message in the status line on the last line of the screen. An older interface, fb_display_message_prompt() is also available to C routines

[8]The saved signal handlers will be reinstalled by the recursively-called *Fabform* when it exits. Since there is only one set of saved handlers for *all* inocations of *Fabform*, very bad things will happen if *Fabform* is not told to install its preferred set of handlers. The FABFORM_SET_SIGNALS bit should be set whenever there is any doubt whether *Fabform*'s preferred set of handlers is installed. The worst thing that can happen by setting the bit is having do a little unnecessary computation.

that leaves the cursor at the end of the message being displayed rather than returning the cursor back to the field it was on.

```
char fb_query_yes_no(prompt)
        char *prompt;

(fabform:query-yes-no prompt)
```

Display the prompt on the last line of the screen and wait for the user to press "y", "Y", "n", "N", or control-G. Returns either 'y', 'n', or '\007' (or the equivalent characters in Lisp).

```
char fb_query_single_keystroke(prompt,validchars)
        char *prompt,*validchars;


(fabform:query-single-keystroke prompt validchars)
```

Display the prompt on the last line of the screen and wait for the user to press a single keystroke. If `validchars` is non-NULL, wait until user presses a keystroke that is in `validchars`.

The Lisp version returns a character object. `nil` may be specified for `validchars`.

```
int fb_query_string_value(prompt,string,allow_spaces)
        char *prompt,*string;
        int allow_spaces;

(fabform:query-string-value prompt allow-spaces)
```

Displays the specified prompt on the last line of the screen and allows the user to enter characters as long as space permits on the bottom line. If `allow_spaces` is non-zero, the user may enter spaces, otherwise, they are ignored. The user's input is returned in `string`, and the return value is the length of the input. A -1 is returned if the user pressed control-G.

The Lisp version returns the string entered by the user or `nil` if control-G was pressed. `allow-spaces` should be specified as `nil` if spaces are not to be allowed.

```
char * fb_get_field_value(field_name,op_name,op_instance)
        char *field_name,*op_name;
        int op_instance;

(fabform:get-field-value field-name op-name op-instance)
```

Return the value of the field named `field_name` within the op_instanceth occurance of operation `op_name`. Returns NULL if the field could not be found.

```
int fb_set_field_value(field_name,op_name,op_instance,
                       field_value,refresh)
        char *field_name,*op_name,*field_value;
        int op_instance,refresh;

(fabform:set-field-value field-name op-name op-instance
 field-value refresh)
```

Sets the value of the field specified by `field_name`, `op_name`, and `op_instance` to `field_value`. No validation is done on the value, though truncation is performed if necessary. Read-only and hidden fields can be modified using `fb_set_field_value()` if desired. If `refresh` is non-zero, the new value will immediately be displayed on-screen, otherwise it will be updated whenever the screen is refreshed. The return value will be 0 if successful, -1 if the specified operation instance could not be found, and -2 if the field could not be found.

```
char ** fb_get_field_restrictions(field_name,op_name,op_instance)

(fabform:get-field-restrictions field-name op-name op-instance)
```

Returns the list of resiricted values associated with a field. Returns `(char *) 0` if the field could not be found or if there are no restrictions on the

field. Each entry in the list is a pointer to a possible value for the field. The last entry in the list is NULL.

The restriction list returned in the current implementation is in fact the list used internally by *Fabform*, thus any changes to that list will affect the validation of entries for the field. In addition, different fields may share a restriction list (eg. via oneof-class declarations), thus changing a list may affect more than one field.

The Lisp implementation returns a list of strings corresponding to the restrictions. There is no sharing of data in the Lisp implementation, thus the returned list and its contents may be mutated without consequences.

```
int fb_set_field_restrictions(field_name,op_name,op_instance,
                              restrictions,free_storage)
        char *field_name,*op_name,**restrictions;
        int op_instance,free_storage;

(fabform:set-field-restrictions field-name op-name op-instance
                                restrictions)
```

All parameters (and the return value) have the same properties as their counterparts in `fb_set_field_value()`. The restriction list specified by `restrictions` should be a list of pointers to C-style strings of possible values for the field and should be terminated by NULL. `free_storage` should be set if the restriction list and all of its entries were allocated via malloc() and should be free()d when the restriction list is no longer needed by this field (ie. when a new restriction list is specified or when this instantiation of *Fabform* exits).

The Lisp implementation takes a list of strings as the `restrictions`. The entries in the list are copied into newly allocated storage before being installed as the restriction list for the field and will be freed when they are no longer needed.

```
int fb_set_field_comment(field_name,op_name,op_instance,comment)
        char *field_name,*op_name,*comment;
        int op_instance;
```

```
(fabform:set-field-comment field-name op-name op-instance
                           comment)
```

Sets the comment printed when the cursor is on a given field. The return value will be 0 if successful, -1 if the specified operation instance could not be found, and -2 if the field could not be found.

```
int fb_place_cursor_on_field(field_name,op_name,op_instance,
                             refresh)
        char *field_name,*op_name;
        int op_instance,refresh;

(fabform:place-cursor-on-field field-name op-name
                               op-instance refresh)
```

Moves the cursor to the specified field. If refresh is non-zero (or non-nil), the screen is updated. Returns 0 if successful, -1 if the operation instance could not be found, -2 if the field could not be found.

```
int fb_delete_region(start_op_name,start_op_instance,end_op_name,
                     end_op_instance,refresh)
        char *start_op_name,*end_op_name;
        int start_op_instance,end_op_instance,refresh;

(fabform:delete-region start-op-name start-op-instance end-op-name
                       end-op-instance refresh)
```

Delete the region of the form beginning with start_op until the end of end_op. If the cursor is on a field that is deleted by the operation, it will be moved to a new field if possible. The region must begin at the start of a template file and the portion of the form following the end of the region must also begin at the start of a template file. If refresh is non-zero (or non-nil), the screen is updated. Returns 0 if successful, -1 if the start operation instance could not be found, -2 if the end operation instance could not be found, and -3 if a bad region is specified.

```
int fb_insert_region_before(op_name,op_instance,param_file,
                        template_dir,refresh)
        char *op_name,*param_file,*template_dir;
        int start_op_instance,refresh;

(fabform:insert-region-before op-name op-instance param-file
                                template-dir refresh)
```

Inserts the region defined by `param_file` into the form just before the specified operation instance. The operation instance must begin at the start of a template file. `template_dir` specifies where to look for template files referenced in `param_file`. Use of globals or oneof-class defnitions in the specified parameter file will not affect fields already existing in the file, thus their use is discouraged. If refresh is non-zero (or non-nil), the screen is updated. Returns 0 if successful, -1 if the operation instance could not be found, -3 if a bad region is specified, and -4 if the parameter file could not be opened. Very bad things can happen if an error occurs while reading the parameter file - usually resulting in the program exitting with an error message.

```
int fb_insert_region_after(op_name,op_instance,param_file,
                        template_dir,refresh)
        char *op_name,*param_file,*template_dir;
        int start_op_instance,refresh;

(fabform:insert-region-after op-name op-instance param-file
                                template-dir refresh)
```

Just like fb_insert_region_before except that the region is inserted just after the speficied operation instance. A bad region error will result if the operation following the specified operation instance does not begin at the start of a template file.

```
int fb_set_ctrlx_keymap(key,function,start_action)
        char key;
struct fabform_retstat (*function)();
        int start_action;
```

```
(fabform:set-ctrlx-keymap key function-handle start-action)
```

Bind the keystroke control-X key to the specified function. key may be any character in the 128 character standard ASCII set. function should be a pointer to a function of the form normally attached to a field (i.e. the same interface for receiving and returning values). start_action is identical to the field that would have been specified as a part of the function definition list passed to *Fabform* upon startup.

In the Lisp interface, function-handle should be an object returned by fabform:make-fabform-function or fabform:defun-fabform-function. Similarly, start-action should be a (possibly null) list of keywords specifying which actions to take before executing the function.

A key binding may be deleted by either defining a new binding or specifying a NULL for function (or nil for function-handle). The application-defined key bindings take precedence over any bindings that *Fabform* uses, thus applications should be careful about how they use bindings. Also, upper and lower case characters are treated differently.

The key bindings are unique to each invocation of *Fabform*. The keymap is reset to it's default state (i.e. no application-defined bindings) when *Fabform* is invoked. Currently the only method of binding keys immediately after *Fabform* is invoked is to specify a function named "fabform_init" in the function definition list given to *Fabform* upon startup. This init function should then set up the keymaps as desired.

```
int fb_set_esc_keymap(key,function,start_action)
        char key;
struct fabform_retstat (*function)();
        int start_action;

(fabform:set-esc-keymap key function-handle start-action)
```

Just like fb_set_ctrlx_keymap (fabform:set-ctrlx-keymap) but binds keystrokes of the form ESC key instead.

```
int fb_get_movement_repeat_count()

(fabform:get-movement-repeat-count)
```

Returns the value of the movement repeat count entered using control-U. Useful for functions that are bound to keystrokes which move the cursor via `fb_place_cursor_on_field`, for example.

```
struct fabform_retstat fb_exec_fabform_function(funcdef,field_value,field_name,
                                                op_name,op_inst,filenam)
    struct fabform_functions funcdef;
    char *field_value,*field_name,*op_name,*filenam;
    int op_inst;
```

Executes a function of the type that is attached to a field and returns the correct value even if the function is coded in Lisp (The action field should have the FABFORM_LISP_CALL bit set if a Lisp function is to be called).

There is currently no Lisp implementation of `exec_fabform_function`.

6 Linking the *Fabform* Subroutine

The structures and values necessary for defining functions are in the include file "/usr/cafe/include/fabform.h". To create the executable file for an application using *Fabform* as a subroutine, simply specify "/usr/cafe/lib/fabform.a" in the list of files to link in. All of the necessary library functions have already been linked into the fabform.a file.

The Lisp implementation lives in a package called "fabform" in "/usr/cafe/lib/fabform.fasl" or "/usr/cafe/lib/fabform.lisp". Lisp applications written under Franz Extended Common Lisp should probably have something like

```
(require 'fabform "/usr/cafe/lib/fabform.fasl")
```

at the start of the application. Remember to specify the .fasl or .lisp suffix or the wrong file will be loaded.

Support has also been added for applications using Kyoto Common Lisp. These applications should load the file "/usr/cafe/lib/fabform.lsp". Applications that may run under either Franz or KCL should load the .lsp file. Do not, under any circumstances, attempt to directly load "/usr/cafe/lib/fabform.o" under KCL. In addition, the KCL implementation requires the presence of "kcl-ffsupport.o" in /usr/cafe/lib. If you are not running in the cafe environment, you will need to obtain this file and install it in an appropriate location.

There are a few precautions that should be taken by applications that link with the *Fabform* routines. Any global symbols of function names beginning with "fb_" or "screenio_" are reserved for use by *Fabform*. If the application attempts to use any globals in this namespace, disasters are likely to happen. In addition, *Fabform* uses parts of the *curses* and *termcap* libraries, thus applications should be wary of any routines or globals used by those libraries.