AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

FE—A Multi-Interface Form System

By R. M. PRICHARD, JR.*

(Manuscript received April 9, 1984)

The Form Editor system provides visual "form" and "menu" capabilities for applications based on the *UNIX*™ operating system. It offers many features usually not found together in other systems that perform similar functions. For example, it includes a program-level interface library, an executable component for data collection, and a multi-hardware/operating system environment. Because the software and language interfaces are simple and require minimal programming background for most applications, a significant reduction in system design and development time is possible. This paper discusses the capabilities and implementation of the Form Editor system.

I. INTRODUCTION

Electronic forms[1] are used in many programming applications to produce software systems with good end-user interfaces. A "form" is an image of a printed document with video attributes or characters representing the locations of required or requested information. Such forms generally fall into one of the following three classes: (1) control, (2) report, or (3) data collection.

Control forms (Fig. 1a), often referred to as control frames[1] or menus,[2] provide processing control by allowing a user to enter a letter or number that corresponds to the displayed list of allowed processing options. The program, on receiving the selected request, can respond in many different ways. For example, it can display a different control menu, report, data-entry form, or message, or prompt the user for additional input.

* AT&T Bell Laboratories.

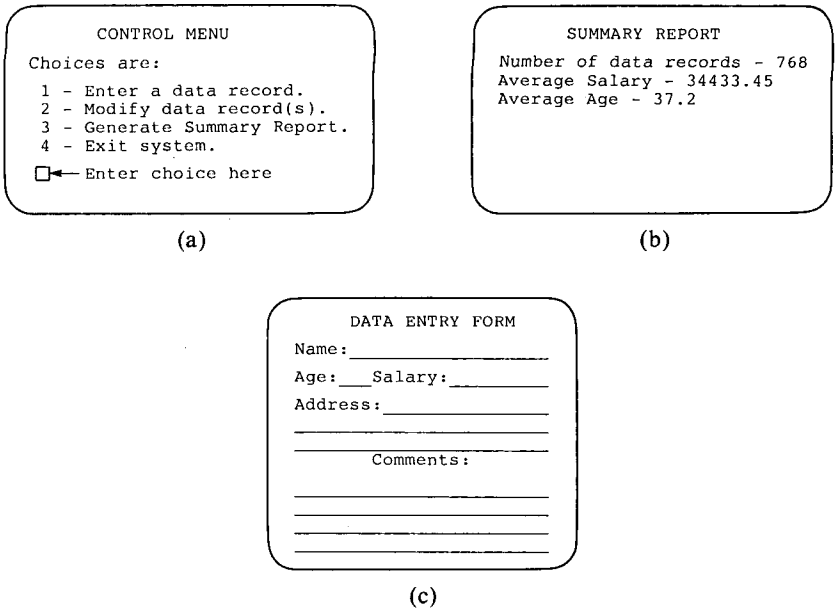

Fig. 1—(a) Sample control form. (b) Sample report form. (c) Sample data collection form.

Report forms (Fig. 1b) can contain descriptive text and data (processed or unprocessed) and are static displays. To change the data on the report, the user changes report input data elsewhere, and regenerates the report.

Data-collection forms (Fig. 1c) can require significantly more interaction by virtue of the amount and types of data that must be entered or changed. These forms can exceed the horizontal and vertical dimensions of the user's terminal and the form fields may need to be randomly accessed, validated, and/or computed. In addition, advanced text editing capabilities such as character insert, delete, or overstrike may be required.

What should be noted at this point is that the capabilities that must be provided by application software to support the possible user interactions with forms can vary significantly. Criteria have been defined for designing forms[3] and several forms-management languages and systems[4–6] have been described. In these reports, emphasis is placed on the generic operations on forms within office-automation systems and the interaction between the system users and the forms is barely addressed.

The FE system was developed as a programming tool for *UNIX* operating systems that need to support form-oriented user interfaces. Though it was initially developed to support data-collection applica-

tions, the system has been used to support all three form classes within an application.

The FE system is easy to integrate into new or existing application software and the form-definition language is simple and supports many capabilities required by form-oriented applications. It consists of an executable system component (Executable FE) and a library of C programming language[7] routines. The library provides terminal screen management and field manipulation routines. Support for CRT as well as non-CRT terminals is provided via the Termcap[8] or Terminfo[9] databases, and "hooks" to standard *UNIX* operating system utility- and application-specific programs exist to provide capabilities not directly supported by the FE system.

II. CURRENT STATUS

The FE system (version 3.1) is currently used in more than 100 applications and is available at most *UNIX* operating system-based computers in AT&T. The software currently runs on Digital Equipment Corporation's VAX* 11/750-780 and AT&T's 3B processor line under AT&T *UNIX* System V or *UNIX* 4.1/4.2 BSD operating systems. It has also recently been installed on the AT&T *UNIX* PC. The FE system (both the library and executable FE) is written in C with the executable component requiring about 120K bytes of memory and the library requiring 85K bytes.

The FE system has been used for a wide variety of applications in AT&T. Most applications (about 80 percent) are developed by individuals who are not software experts, but engineers, secretaries, and managers with some familiarity of the *UNIX* operating system. The complexity of these applications varies greatly. For example, several recruiting and professional society membership databases are applications that require only FE and a few shell program filters, whereas an automatic test program generator for integrated circuits integrates FE with many other system components. It has also been used as a report generator and to provide simple spread sheet-like capabilities for several applications.

The FE library has been used for those applications that require rigid control over system processing and/or data collection. Examples of typical applications that currently use the library include a field-oriented visual editor, front ends for several database management systems, and a budget management system. The library has been integrated into several commercial systems currently under development and is being considered as a standard software development tool in several AT&T Bell Laboratories computing centers.

* VAX is a trademark of Digital Equipment Corporation.

III. THE FE SYSTEM ARCHITECTURE

The executable `FE` system module operates as an editor in which the unit of reference is a form instead of a line or record in text editors such as `ed` or `vi`. Instead of inserting or deleting records in the data file and changing characters in a record, the user inserts or deletes forms and changes the fields contained on a form. Each form is described in a Physical Form Definition File (PFDF) and sets of PFDFs can be combined or concatenated by Logical Form Definitions (LFDs).

The executable module actually consists of two executable programs, as shown in Fig. 2. `FE` provides the end user with the capabilities provided by the `FE` library routines and the `FEio` module provides the underlying database management functions.

`FE` is usually invoked from within a shell program that passes the name of the data file, the path of the PFDF directory, the name of a file that contains the LFDs, and other optional arguments. These arguments can be used to custom tailor the `FE` system for an application. `FE` internally activates the `FEio` module and establishes co-process communication using pipes over which simple commands are sent to instruct `FEio` to perform such functions as data searching, retrieval, and update. `FEio` responds to these commands by returning a command status or one or more data records.

There are two advantages to this type of interface. An application can provide its own data interface program when the format of the `FEio` data file is inappropriate, and `FEio` can be used as a concurrent process by other application software components to provide a direct-access interface to executable `FE` data files.

The executable component of the `FE` system functions much like a filing clerk with the responsibility of maintaining the forms in one or more filing cabinet drawers (`FE` data files). For example, executable can be instructed to place a new form at a specific location in the file or search the data file for a form to be "pulled" for review or changes.

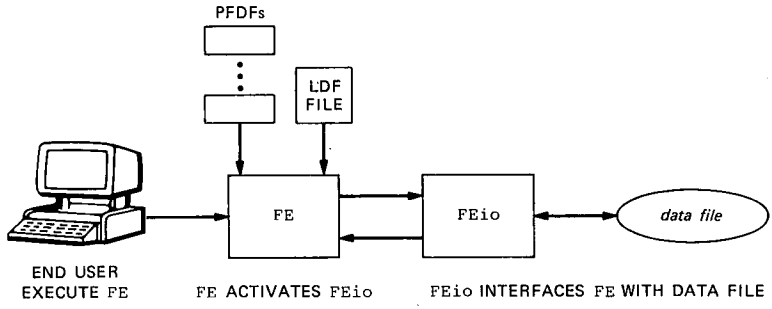

Fig. 2—End user, `FE`, and `FEio` interrelationship.

The FE data file can contain many different types of forms (logical forms), each of which may be referenced by its relative location from the front, or top, of the file or by its associated name. As forms are inserted into the data file, the size of the file increases; as they are deleted the size decreases.

IV. LIMITATIONS

FEio functions much like any text editor. It creates a temporary file to which modifications are applied and builds an internal index that points to the file locations of the forms each time it is invoked. This initialization time grows linearly with the size of the FE data file. However, once past the initialization step, the response time for most commands (including interform searching) is usually within three to six seconds. Editing data files that are close to the host system's file size limitation (usually 1 to 2 megabytes) can cause file overflow problems. When this happens, FE attempts to terminate processing as gracefully as possible.

V. THE FE SYSTEM FORM-DEFINITION LANGUAGE

Each form used in an application requires the creation of a PFDF that contains a description of the form's template and the attributes of its associated fields written in FE's high-level form definition language. These form-definition files can be concatenated or "pasted" together at run time to form "logical forms." This logical form capability provides the application developer with the ability to treat PFDFs as modular constructs that can be used as building blocks for form definition. For example, if two distinct application forms (Fig. 3) contained a common section, PFDF "A" could define the common section and two other PFDFs ("B" and "C") could define the unique sections. Two LFDs specifying how the FE system is to "bind" the form components together would then be defined.

5.1 Dynamic form construction

The internal representations of forms defined for an application are not static in the sense that they must be compiled along with the FE source code. Instead, each form is translated and loaded into memory at run time where it remains until a request to load a different form is received.

This loading strategy provides the following benefits:

1. Logical forms sharing a common PFDF will always be constructed correctly.
2. PFDFs may be modified during run time to provide "dynamic" forms.

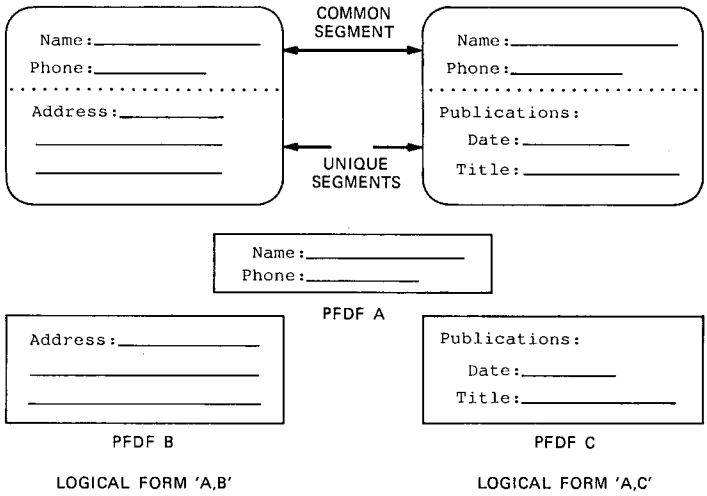

Fig. 3—Logical forms example.

3. Memory requirements are greatly reduced for multi-user, multi-form applications.

The major drawback associated with this strategy is that the time required to translate and load a form into memory is directly proportional to the size of the form. This translates to about two or three seconds for a form 24 lines or fewer and up to 20 seconds or more for forms larger than 100 lines. However, the forms in most applications seem to be in the 20- to 30-line range and two or three seconds seems to be acceptable. For applications that frequently switch between many large forms, the overhead may be unacceptable.

5.2 Physical form definition files

PFDFs are ASCII data files that can be created with any text editor, e.g., `ed`, `vi`, etc., and may define an entire form, or a form segment common to many forms. Because PFDFs are simple text files and are translated and loaded at run time, they can also be created during run time by application software. For example, an `FE`/Database Management System prototype exists that dynamically creates PFDFs from the database definition (transparent to the end user), and allows the user to input, modify, and query the database using the executable component of the `FE` system.

Each PFDF consists of three sections that describe the associated form's template, field attributes, and field help text. Only the template section is mandatory; an application could omit all other PFDF sections and default attributes would be assigned for all fields. Figure 4 shows the organization of these three sections on a typical PFDF.

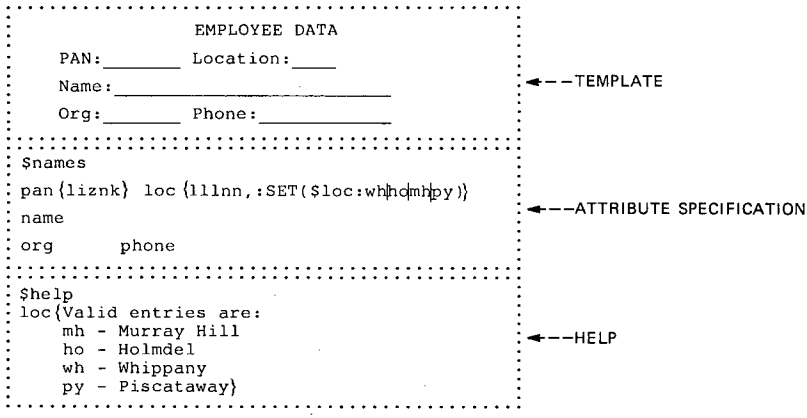

Fig. 4—PFDF format and organization.

The "Template" section describes the form image that is displayed on the terminal. Field locations (indicated by a series of contiguous underscore characters), lengths (the number of contiguous "_"s), and simple template text are elements of this section.

The "Attribute Specification" section of a PFDF is used to assign attributes to the fields that appear in the template section. Each field can be defined in terms of a name (*Fname*), data attributes (*Att_tuple*), a value assignment expression (*Asg_expr*) and a validation rule (*Valid_rule*) according to the following syntax:

Fname{*Att_tuple*,=*Asg_expr*,:*Valid_rule*}

Fname consists of a string of characters taken from the character set {A-z,0-9_-,.}. *Att_tuple* is a coded five tuple that defines the following field attributes:

1. Field enhancement attribute (underline, inverse video, red, blue, etc.).
2. Data type (integer, numeric, upper case, dollar, etc.).
3. Justification (right, left, centered, zero-fill).
4. Constant/Default/Locked/Shared/Protected value.
5. Key field (for interform searching).

For example, the first field of the form shown in Fig. 3 has *Fname* declared as "pan," its associated attribute tuple as "liznk." This attribute tuple decodes as follows:

'l'-Underlined on CRT,
'i'-Integer field,
'z'-Zero filled,
'n'-No protection,
'k'-An interform search key field.

The third field on the same PFDF declares the *Fname* attribute as "name" and allows the FE system to assume system defaults for all others.

Field values can be defined with an algebraic expression in *Asg_expr* that can include references to numeric constants, other fields, external programs, and shell-exported variables instead of end user input. For example,

```
total{1nrcn,=($field1 + ENV(value))/2}
date{1clcn,=EXT(date)}
```

The first example defines the value of field `total` as one half the value of the sum of field `field1` and the value of the shell environment variable `value` (as specified by the built-in function `ENV()`). In the second example, the value of field `date` is to be filled with the results of the external program `date` (as specified by the built-in function `EXT()`). The only requirement of programs executed by `EXT()` is that they return their result(s) to the standard output device. `EXT()` gives the application developer a direct "hook" to the *UNIX* system environment to obtain computation or validation capabilities not directly available in the form definition language.

A data-validation rule *Valid_rule* can be provided in each field-attribute specification. The syntax of the validation rules is algebraic, with arithmetic and logical operators taken from the C programming language. Validation rules specify under what conditions the field is logically correct or valid. Rules may contain references to other fields, external functions, constants, and environment variables. In addition, validation rules may also contain arithmetic operators. Below are two examples of validation specifications for character and numeric type fields:

(1) `loc{1llnn,:SET($loc:mh|ho|wb|wh|py|cb)}`
(2) `sum{1nlnn,:$sum >= $val1 && RANGE($sum:0-$val2)}`

Example 1 specifies that the value of the field `''loc''` must be a member of the set `mh`, `ho`, `wb`, `py`, or `cb` to be considered correct. `SET()` is the built-in function used for specifying a set of strings. In (2), the rule specifies that field `sum` must be greater than or equal to the value of field `val1` and also greater than or equal to zero and less than or equal to the value of field `val2`. Since `SET` and `RANGE` return Boolean results, the unary operator `!` (not) can be used to indicate "not a member of a set" (`!SET()`) or "outside the range" (`!RANGE()`).

A robust set of built-in functions exists for both value assignment and validation-rule specification that includes vector operations and regular-expression pattern matching.

The "help" section of a PFDF is used to define field-specific "help"

text for any or all fields defined in the Attribute Specification (`$names`) Section. The user may request the display of a field's "help" text by positioning the cursor on the selected field and entering the appropriate command.

In addition to explicitly defining the "help" text for a field in this section of the PFDF, the form designer can specify that the text source be retrieved from a data file or generated by an application program.

5.3 Logical form definitions

PFDFs may be combined, or concatenated by LFDs. LFDs can specify a simple or hierarchical concatenation of PFDFs. For example,

```
(1) payroll
(2) payroll,work_hist
(3) {dept{3*group{10*members}}}
```

Logical form (1) is defined as a single copy of PFDF `payroll`, while (2) defines a form that is constructed using a `payroll` PFDF concatenated with a `work_hist` PFDF. Example (3) is a hierarchical definition that translates as follows: concatenate one PFDF `dept` form with three PFDF `group` forms, each of which is concatenated with ten PFDF `members` forms.

VI. END-USER INTERFACE LANGUAGE

The end-user language is partitioned into "off-form" and "on-form" commands. "Off-form" commands are functionally and syntactically similar to the language of the text editor **ed** and operate on forms as a unit; for example, they delete or append a form. Other data file manipulation commands include multi-key sorting, searching, data extraction, and hard copy reproduction of forms, in addition to standard editing commands.

The on-form commands* are modeled on the visual editor **vi**, and apply only after an off-form command is entered to display either a blank form for input, or a filled-in form from the data file. These commands stay in effect until the user indicates the end of input or changes for that form. When the cursor is positioned on a form, the end user can be in either a "positioning" or a "data-change" mode.

In the "positioning" mode, single keystroke directional commands can move the cursor forward, back, up, or down. The cursor may also be positioned to a field via regular expression pattern matching and direct or relative line addressing. For example, move to form line *N*, or move up(−) or down(+) *N* form lines. When in the data-change mode, any characters entered by the user are placed on the field where

* Executable **FE** uses the data-collection library routine **FE_edit**.

the cursor is positioned. The user may enter or exit the positioning or data-change mode at any time during the editing session by typing the appropriate command.

Field editing capabilities include "insert character," "delete character," "erase field," "replace characters," and others; each of these can easily be "turned off" or redefined if considered inappropriate for the application.

VII. DATA FILE ORGANIZATION AND FORMAT

The organization of an **FE** data file can be seen in Fig. 5.

"Header" records delimit logical groups of associated "name/value" pair records. Each header record contains the following four components (separated by tab characters):

1. The logical name of the form,
2. The values of all fields defined with the "interform search" attribute,
3. The logical form's PFDF description, and
4. The number of errors contained in the associated data.

Name/value pair records are organized to match the layout of each logical form definition. For example, Fig. 5 represents the output file organization for the PFDF shown in Fig. 4. Changes made to a PFDF (adding, deleting, or changing the field order) do not require an **FE** data file to be changed. This feature is important for applications with volatile data input and validations requirements, since the field data will always "map" to the correct position on the constructed form.

Because **FE** data files are stored in ASCII format, simple **sed**, **awk**, or C programs can be written to generate reports or massage the data to a format acceptable to other application components. For applications that are already name/value pair oriented, the "data file"/"application system component" interface may be minimal.

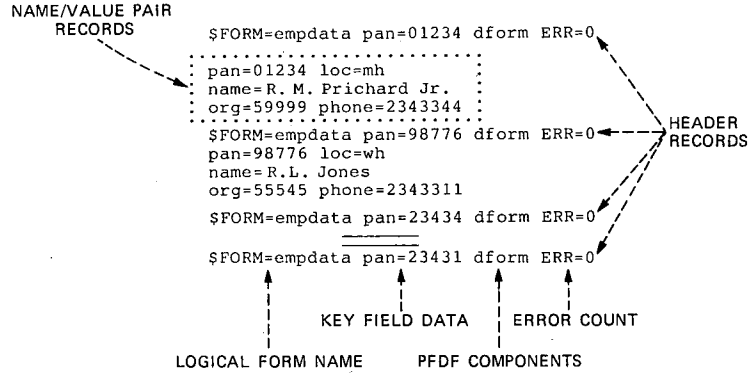

Fig. 5—**FE** data file organization and format.

VIII. EXT() BUILT-IN EXAMPLE

The power and versatility of `EXT()` can be seen in the following example:

A query form (Fig. 6) is displayed with only the entry of a single field "name" allowed. All other field values are supplied by a database retrieval program that uses the value of the "name" field as a search key for extracting the remaining field values from the associated databases.

This application can be implemented by defining a facsimile of the retrieval form in a PFDF with appropriate field names. One field's assignment attribute invokes the retrieval program as follows:

```
snn{Att_tuple,=EXT(ret_prog $name)}
```

To execute the retrieval program to fill in the rest of the form, the user would enter the appropriate data in the name field and then enter the **FE** system command "compute". `Ret_prog` would then extract the remaining form data from the database and write each field's name and associated value to the standard output device. For example,

```
pan=098999
ssn=212-22-2323
org=59999
  :
  :
```

The **FE** system would then read the transmitted data and update, on the terminal, the field values as they are received.

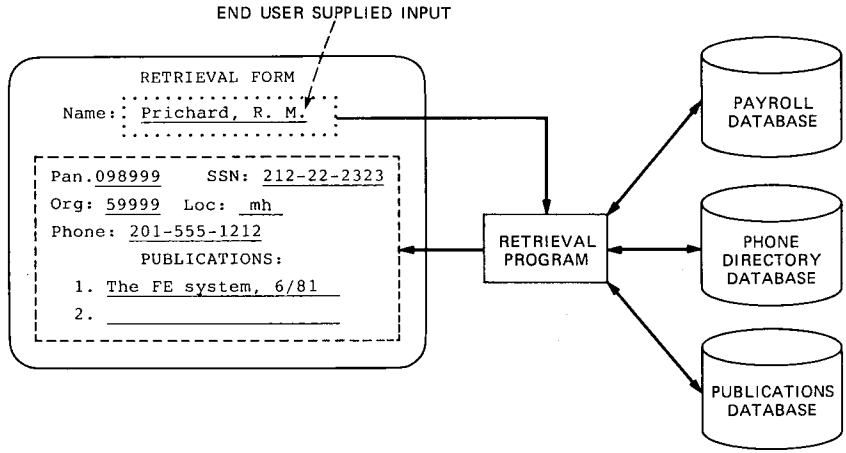

Fig. 6—Using the `EXT()` build-in.

IX. FE FORM LIBRARY

The form library currently consists of 80 subroutines. Of these subroutines, 25 are high-level routines intended for application interfaces and provide the following functions:

1. Virtual terminal and library initialization.
2. Physical form translation and loading.
3. Program placement of field values on forms.
4. Program retrieval of field values on forms.
5. End user input, modification, and validation of field values.
6. Termination or "wrap up" processing.

At the core of the FE library (Fig. 7) reside the virtual terminal interface and low-level form manipulation routines. The virtual terminal interface uses either the Termcap or Terminfo database and utility routines (based on the host operating system) to provide support for CRT as well as non-CRT devices.

The terminal interface and form manipulation routines are in turn used to form the application interface routines. The lower-level routines are intended for use by applications that require access to FE's internal structures or terminal control sequences.

The following example shows how the high-level library routines can be used to collect data via a form and "dump" the field names and data values to standard output in a name/value pair format.

> NOTE: Regular type represents the library routines while *italicized* type represents global library structures and variables.

```
#include <stdio.h>
#include "FE_structs.h"

#define REDRAW 1
#define FULLSCREEN 1

main()
{
```

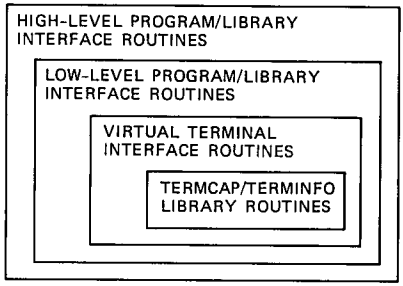

Fig. 7—FE library architecture.

```
(1)    extern int _dlinecnt, _fldcnt[];

(2)    extern struct symbol **_cp[];

       char buff[512];
       int i,j;
       struct symbol *cp;

(3)    FE_init(open("/dev/tty",2),FULLSCREEN);
       strcpy(buff,"dataform");

(4)    FE_bldform(buff,"/PFDF/form/directory");
(5)    FE_edit('c',REDRAW);

(6)    for(i=0;i<_dlinecnt;i++)
         for(j=0;j<_fldcnt[i];j++)
         {
           cp=_cp[i][j];
           printf("%s=%s\n",sp->name,sp->sval);
         }
     }
```

In this sample program, (1) and (2) declare two global FE library integer variables *_dlinecnt* (data line count) and *_fldcnt[]* (data line field count). These two variables define the dimensionality of the cursor positioning matrix *_cp[][]*. Each element of *_cp* is a pointer to the symbol table element associated with a single field on the current form in memory.

The first FE library routine called FE_init initializes the terminal interface routines (3). Two arguments are passed to it: a file descriptor for ''/dev/tty'' (opened for read/write) and FULLSCREEN to indicate that the full screen is to be used (not adjusted by the terminal's output baud rate). To build a form in memory the FE_bldform is called (4) with the logical form definition (stored in "buff," argument 1) and the directory where the logical form's PFDF components reside (argument 2).

To collect the data, FE_edit (5) is called with arguments 'c' and REDRAW. These arguments will cause FE_edit() first to redraw the template on the CRT and then to enter the "change mode," which allows the user to enter data immediately. This library routine provides the end user with a vi editor-like interface for entry and modification of the data. When the data entry session for the form is completed (the end user enters the appropriate "quit" command), control is returned to the program.

At (6), each element of the cursor positioning matrix *_cp* is "visited" and the associated field names and values are written to the standard output device via a printf.

Though this example uses the global internal structures of the library, the code in (6) can be replaced by the library's "get field value by name" routine calls. For example, if the *Fname* attribute of a field on the form built by the call to `FE_bldform` is `department`, the associated value could be retrieved by calling the `FE_get` routine as follows:

```
FE_get(''department'',value,changed,'r');
```

`Value` defines the string address where the retrieved value of the field `department` is stored, and `changed` is a string address where `changed[0]` indicates if the value of the field has (value = 1) or has not (value = 0) been changed. The fourth argument instructs `FE_get` to reset the associated field's internal "changed" flag to zero.

X. OBSERVATIONS

User community feedback has been favorable since the initial **FE** system prototype was released. A brief summary follows:

1. The end-user language is easily learned by users already familiar with the *UNIX* operating system because of the functional and syntactical similarity with the standard text editors **ed** and **vi**. For individuals not familiar with text editors, the time required to learn the **FE** end-user language is no more than that required to learn any other editor.
2. System documentation is generally acceptable.
3. Defining a form does not require much time because the form-definition language is simple.
4. Interfacing with most application components is not difficult.
5. Significant savings in design and development time have been achieved for all applications.

The last observation (5) can be best illustrated with an example.

An application system required about 12 to 15 weeks to develop a prompting program that only provided an initial data input facility. A total of eight different forms were supported and the software provided basic intra-form validation and a crash recovery capability. Data modification was implemented with the standard system text editor. When a new release of the system was built, **FE** was used to provide data collection and modification capabilities for more than 35 forms. Data validation was provided by application supplied software as **FE** validation was not yet available. The entire process to integrate **FE** into the application required less than two weeks to define the forms and build a simple software filter to interface the data file with other application components.

Because the form definitions were volatile in the initial development phases of the project, an even greater savings of development time

was realized since there were no programs to modify and recompile each time there was a form-definition change; all that was required was for the PFDFs to be changed using a text editor.

I believe the success of the FE system can be mainly attributed to the form-definition language (PFDs and LFDs) and the ability to support data entry as well as report and processing-control data forms. In addition, software developers have found it easy to "custom tailor" both the libraries and executable FE to meet the unique requirements of their applications.

In order for "screen" management software to exist in the future, generalized interfaces should be developed to support alternate input devices such as "mice," touch sensitive screens, etc. Though nothing will replace the standard keyboard for input of textual data, these alternate input devices will find their calling in processing control and cursor positioning.

XI. ACKNOWLEDGMENTS

I am indebted to D. G. Korn, K.-P. Vo, and other colleagues for their technical inputs and the FE system user community for providing constructive feedback.

REFERENCES

1. K.-P. Vo, "IFS—A Tool to Build Integrated, Interactive Application Software," AT&T Tech. J., this issue.
2. J. W. Brown, "Controlling the Complexity of Menu Networks," Commun. ACM, *25,* No. 7 (July 1982), pp. 412–18.
3. D. V. Morland, "Human Factors Guidelines for Terminal Interface Design," Commun. ACM, *26,* No. 7 (July 1983), pp. 484–94.
4. D. Tsichritzis, "Form Management," Commun. ACM, *25,* No. 7 (July 1982), pp. 45–78.
5. P. De Jong and R. Byrd, "Intelligent Forms Creation in the System for Business Automation(SBA)," IBM Research Report RC 8529, 1980.
6. D. W. Embley, "A Form Based Non-procedural Programming System," Technical Report, Department of Computer Science, University of Nebraska, 1980.
7. B. W. Kernighan and D. M. Ritchie, "The C Programming Language," Englewood Cliffs, N. J.: Prentice Hall, 1978.
8. W. N. Joy and M. R. Horton, "UNIX User's Manual 4.1 BSD," TERMCAP(5) manual page.
9. M. R. Horton, "UNIX System V Release 2.0 Programmer Reference Manual," TERMINFO(4) manual page.

AUTHOR

Reuben M. Prichard, Jr., B.S. (Computer Science), 1976, and M.S. (Computer Science), 1978, Rutgers—The State University; AT&T Bell Laboratories, 1967—. Mr. Prichard has been a member of the Business Analysis Systems Center since 1976, where his work involves the design and development of decision support systems and software tools to support that work. His research interests include software design methodologies and human factors.