A Sophisticated Microcomputer User Interface

Richard Snodgrass

Carnegie-Mellon University
Department of Computer Science
Pittsburgh, PA 15213

Abstract

The design and implementation of a menu-oriented interface for personal computers is discussed. Factors pertaining to the cognitive limitations of users are examined and their impact on the design of the system is described. The major attributes of the system are (1) all communication between the operator and the computer is through menus or forms (which are analogous to hard copy documents); (2) extensive help is available at all times; (3) the interface can adapt to the experience of the user; (4) the display processing time is short; and (5) an external data format exists that completely defines the interface. The various components of the interface are discussed in detail, followed by a discussion of the implementation.

1. Introduction

Although some research has been done on developing intelligent user interfaces for large systems [2, 12, 14, 16, 20, 22], very little investigation has focused on the adaption of these interfaces to a personal computer (also referred to in this paper as a microcomputer system). One reason is that microcomputers impose a stringent set of restrictions on the resources available when implementing or executing a program. These restrictions concern the lack of a virtual address space, the slowness and small capacity of the disk, and the low processing rate. They impose secondary restrictions, such as a small symbol table in the compiler (due to an excessive swapping overhead) which are often more limiting than the obvious deficiencies. Dealing with these constraints severely complicates the design of sophisticated user interfaces and makes research in this area difficult. Another reason for the lack of research concerning microcomputer user interfaces is that personal computers have only recently advanced to the point where effective interfaces are possible.

In spite of the many restrictions that are present in microcomputer systems, personal computers also have some attractive properties not shared by large systems. A personal computer usually has a dedicated video display which can contend with very high data transfer rates (1000 characters per second is typical). Also, although the processing power is not overwhelming, it *is* uniform, whereas users of large multiprocessing systems get the use of a faster CPU only for short periods of time at irregular intervals, which sometimes results in annoying behavior [3]. A third advantage is the availability of low-level primitives of the operating system and of facilities provided by the bare machine. Personal computers require less complex operating systems than conventional computers, since multiprogramming issues (protection, resource sharing, synchronization) are not present. Thus, microcomputer operating systems offer lower overheads and direct access to the resources of the system [11].

This paper describes a user interface which exploits the positive features of a personal computer while dealing effectively with its

limitations. The system is designed to be employed with any application program which requires a substantial amount of interaction with the user, due to an extensive set of available commands, a large amount of data entry and display, or both. In particular, packages currently marketed for small businesses that handle accounting, data base, and/or management functions are prime candidates for the approach examined here. As hardware becomes less expensive, the increased facilities available in a personal computer will allow the general system to be extended to incorporate more knowledge concerning the user and the environment provided by the application program. Thus, it is appropriate to attempt to provide user interfaces for present microcomputers, even though more powerful systems will be available in the future.

2. Design Criteria

There are a number of design criteria which must be met by a user interface [1, 10], whether on a personal computer or on a large system. A particularly important aspect is the nature of the constraints of the user's cognitive processing. People are heavily constrained by the amount of information that they can consider at one time and the length of time that that information can be held in short term memory (STM). Hence, the information available from the system should be simple enough to be quickly and easily assimilated. The system should also be fast enough so that the user's thinking processes will not be impaired by the loss of information in STM due to delays caused by slow response times. An additional constraint is that the retrieval process from long term memory (LTM) is also subject to decay over time. Therefore, the user should be able to get help from the system at *any* time. This information should be available at the place the help is needed, and should refer to the specific context the user was in when help was requested [8]. The system should also exhibit *uniformity*: similar commands at different times should invoke similar actions, in order to reduce the state-dependency of commands. The actions invoked by a particular command should be explicit and unambiguous, resulting in a *transparent* system. Such a system is closely matched to the user's perception of the system, requiring less mental work [21].

A second consideration which is important in the design of interactive systems is the experience of the user. Typical users of these systems vary widely in both the degree of familiarity and the motivation to learn about the system. Many jobs which involve using a microcomputer have large turnover rates, so it is important for the system to cater to novice users. It is important, however, not to ignore the expert user. The interface should be able to adapt its characteristics to the user's experience. How to imbed this ability in the interface is itself a hard problem [7, 17]; at the very least the system should be able to tailor the amount of explanatory text to the desires of the user. Thus the novice would be given a significant amount of guidance by the system, whereas the expert would be presented with a very fast interface that specified a minimum amount of detail.

The remainder of the design criteria for a user interface stem from the character of the communication as perceived by the user. The interface should be *robust*, in that it responds effectively and unambiguously to *any* input, allows the user to recover from simple errors, and discourages illegal input. The user should have a simple means of giving commands to the system, to both reduce errors and increase the command input rate. And finally, control over all aspects of the system must appear to belong to the user.

Although it may seem reasonable to expect all user interfaces to adhere to the criteria given above, very few existing systems meet even a substantial subset of them. The system described here meets all of the listed criteria and can be implemented and run efficiently on a microcomputer system.

3. Basic Design

The design is based on the ZOG system [15, 18], which in turn was based on the PROMIS system [9, 23]. In the ZOG system communication is via menu selection on display terminals with special touch-sensitive screens. A ZOG menu (or *frame*) is simply a list of commands with explanations; each command is associated with an area on the screen which the user touches to select that command. The result of the selection is

another menu with further selections. The network of menus is very large (on the order of 35,000 frames in the PROMIS frame library) so that all communication is by this means. Thus, communication from user to computer is by the discrete selection of semantically meaningful options, and from computer to user by the presentation of information contained in frames. The distinguishing characteristics of ZOG are that the response time for the next display is essentially instantaneous (ZOG is targeted for 0.05 seconds 70% of the time) and that the total set of frames is very large. This coupling of rapid response and a large network produces a qualitatively different user-computer communication philosophy from a standard menu selection scheme, of which there are many [13, 16]. Unfortunately such a system requires very expensive hardware, in the form of touch sensitive terminals, fast disks, and powerful processors. The approach taken in the system described here was to severely limit the knowledge domain (thus lowering by two to three orders of magnitude the number of frames necessary to cover the domain), to relax the stringent time constraints, to introduce verbosity levels in the menus, and to propose an additional object, analogous to a hard copy document, called a *form*, which is more appropriate for performing large amounts of data entry and display. Thus, the system only approximates ZOG in its operation, but the advantages of being able to implement it on a personal computer make it a viable direction in which to pursue user interfaces.

There are five major attributes that characterize the user interface. First, all communication between the operator and the computer is through menus if a simple command is adequate, or through forms if a more extensive data entry and display capability is needed. Second, extensive help is available at all times. This help ranges from a single message concerning a particular item in a form to an entire subnet providing information on a an aspect of the system. Third, the interface can adapt to the experience of the operator by limiting or expanding the amount of information presented by a menu. Fourth, the display processing time is short. And lastly, an external data format exists that completely defines the interface (except for the connections between the frames, either menus or forms, in the network, since arbitrary processing by the application program is possible any time a selection is made or data entered into a form).

These attributes preserve the advantages inherent in menu selection schemes. It is possible for users who are totally unfamiliar with the program or the interface to use the system. Menu selection allows the user to parlay basic knowledge about how menu selection works into detailed knowledge of the system. Also, by locating the relevant knowledge at the site of action, menu selection eliminates the search for this information. However, menu selection has two disadvantages. First, in typical implementations, it is slow. Second, the user is forced to be shown a large amount of explanatory text, which is annoying to the expert user who already knows the information the menu contains. ZOG removes both of these disadvantages with its rapid response time, since the expert user can simply ignore the explanation. The system presented in this paper avoids the disadvantages by exhibiting a moderately fast reponse time (the fastest the hardware can support) and by providing each menu with several versions, each containing more or less detail (ranging from one line to several hundred lines of text). Thus, the attributes described above, when taken as a whole, result in a powerful user interface.

The top-level view of the system is illustrated in Figure 1. The entire user interface is implemented by the *Interface Module*, and all manipulations of the data base are handled by the *Database Module*. Information is transfered between the database and interface modules, and commands are transfered between the interface module and the application program. In the sections that follow, a more precise characterization of menus and forms will be given, followed by a discussion of the implementation on a personal computer and a comparison of the actual performance of the interface with the design criteria specified in the previous section.

4. Menus

A *menu* is a list of commands with explanations. A selection is made by typing the command, thus avoiding the need for touch-sensitive terminals. There are three levels of information associated

with each menu. The *abbreviated* version is a sma. number of lines long, of which only one is displayed at any time. An example (see the appendix for details) is shown in Figure 2a. This version is used by experienced users who want the maximum speed (less than 1 second per menu in the implementation described below) and are familiar with the commands in a particular region of the frame network. The *standard* version occupies the entire screen. The top line is constructed by the

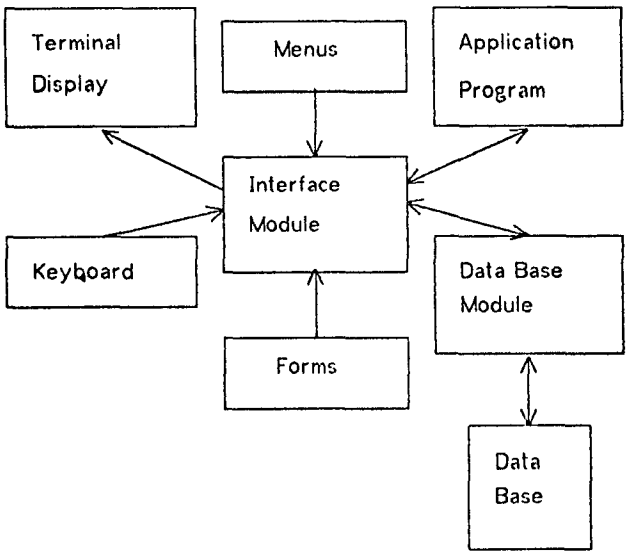

Figure 1: The Top Level Design

system, and consists of the menu name, the system name, and the date. The remainder of the screen contains a listing of the commands along with explanatory text. The same menu might have the standard version shown in Figure 2b.

The third version, the *detailed* version, consists of one or more complete screens of text going into more detail about the subject the menu pertains to and the use of the commands in the menu.

There are two types of commands associated with each menu. The *generic* commands apply uniformly to all menus. The *help* command causes the next version to be displayed (i.e., the standard version when viewing the abbreviated version and the detailed version when viewing the standard version). When the user indicates to the system that she has seen enough of the detailed version, the system returns to the standard menu. A '?' command is equivalent to the *help* command. When viewing the standard version, the *less-detail* command causes the abbreviated version to be displayed. The *quit* command also is allowed within any menu.

Specific commands can be different for every menu, and are specified along with the text of the menus. In the menu described above, the *append*, *edit*, *remove*, and *search* commands are specific commands; the rest are generic. Unique abbreviations of all commands are allowed (one character is usually sufficient to uniquely identify a command).

5. Forms

While menus are appropriate for communicating information which resides in the frame network, and for giving the system simple commands via selections, they are not appropriate for the entry or display of large amounts of data. Instead, the system employs *forms*, which consist of a set of *fields* embedded in uninterpreted text. A field refers to consecutive character locations on the screen and is described by four attributes. The *picture* attribute describes the syntactic form of the information that resides in the field and is used during both data entry and display. The *explanation* attribute is a character string used when the user requests help from the system. The *default* attribute is used when entering data into the form. Finally, the *field name* attribute is used in the interface between the form handler and the database control system.

Forms were designed to exploit the user's intuition regarding data entry and display. Menus could have been used instead for these functions, with one field per menu. However, the context provided by the other information in the form is important, and this context is lost if the fields are spread across several menus. For example, if there is a record in the data base for each employee of a company, then one menu might be used to select the particular employee, another menu would give his skills, and a third menu would give his formal training. If all of this information was displayed at the same time, the user could get a much better idea of, say, whether the employee was qualified for a promotion. Also, forms are appropriate for network data bases (see section 7); other

```
Customer List: A(ppend) E(dit) R(emove) S(earch) H(elp) Q(uit) ->
```

(a)

```
Customer List                    Example System                    April 13, 1980

A(ppend)        This command allows you to add new customers to the Customer
                List.

E(dit)          This command ...

H(elp), ?       This command tells you more about the use of the Customer
                List and the commands that allow you to modify or examine
                this list.

Q(uit)          This command returns you to the main menu for the system.

Command ->
```

(b)

Figure 2: Sample Abbreviated and Standard Versions

approaches might be more suitable for relational data bases [4].

The use of a form for data entry is as follows. First, the text in the form is displayed (in half-intensity if possible). Then the cursor is positioned at the beginning of the desired field and a *mask* is displayed (either in full intensity, or, if the terminal does not support half-intensity, then in inverse video or in some other form of highlighting). The mask is identical to the picture, except that a distinguished character (such as '-') appears everywhere that a user supplied character can go.

If the ENTER key is typed, then the default is entered into the field. Any time a '?' is typed, the explanation string is displayed on the bottom line of the terminal. The picture, which is essentially a COBOL PICTURE [5] with an altered semantics to allow for interactive input, is used to control the characters which will be accepted from the user. Each character position in the field is controlled by a character in the picture. The *edit characters* determine which input characters are acceptable; the edit characters are A (alphabetic), B (boolean, e.g., 't' for true and 'f' for false), X (alphanumeric), and 9 (numeric). All other characters are noise, and are automatically printed when they are encountered and backed over when processing rubouts. The mask consists of the picture with the distinguished character replacing all the edit characters. The picture '(999) 999-9999 Ext. 9999' results in a mask '(---) -------- Ext. ----', and if the characters '1234567' were typed, the field would look like '(123) 456-7--- Ext. ----', with the system expecting seven more digits. When a value is displayed in a field, the successive characters of the value replace the edit characters in the picture attribute.

6. Evaluation

It is appropriate to review briefly the original design criteria to illustrate the extent to which they have been met. The system is friendly, and the user can obtain help from the system at any time. With careful menu design, the information can be kept simple enough to be easily understood. The various levels of menus allow the system to cater to the user's preference of amount of detail. The system will ignore erroneous input, giving an error message in the case of an illegal menu selection or refusing to accept an illegal character during data entry. The system exhibits uniformity in that the generic commands always apply and a '?' causes the system to respond with an explanatory message. The system is fast since it requires at most one disk access to display a menu (see section 7) and since the display can proceed at the maximum rate allowable by the terminal. The user has a simple means of giving commands to the system by selection from the set offered by the current menu. And finally, the user feels that she is always in control of the system (although the system may constrain the range of options available

at any point) since the system is always waiting for a command (or data) from the user.

7. Implementation

The severe limitations imposed by implementing the system on a microcomputer necessitate a careful partitioning of the information which resides in memory and the information which resides on secondary storage. Putting all the information in main memory is infeasible due to the large number of menus and forms in the frame network; putting all of the information on disk results in unacceptable response times. In addition, it is necessary to design the interface module so that the complexity of handling help requests and of dealing with erroneous input is embedded within the module, where it can be dealt with, rather than forcing the application program to handle these conditions. In view of these considerations, the implementation was partitioned into three components: a descriptor file, a preprocessor, and the interface module. The interactions between these three components is illustrated in Figure 3 and examples of the various files are given in the appendix. Although this particular implementation was done using the UCSD Pascal system [19] on a Z-80 microcomputer, the design presented in this paper can be usefully applied on most microcomputer operating systems and languages.

The descriptor file completely defines each frame used by the system. Included in the descriptor file is the text for each frame, the commands associated with each menu (and their Pascal names, which are used by the application program to refer to the command), and the attributes for each field in each form. There are no a priori limitations on the number of commands, menus, forms, or fields that can be declared in a descriptor file. Although the descriptor file is in a simple human-readable format that is self-documenting, the format is designed to be efficiently processed by the interface module. Each menu is represented by a command list, the abbreviated version, the standard version, and the detailed version. Each form is represented by a backdrop containing the text of the form, with consecutive '@'s wherever a field belongs, and a list of attribute tuples, one for each field. All but the command list are read by the interface module

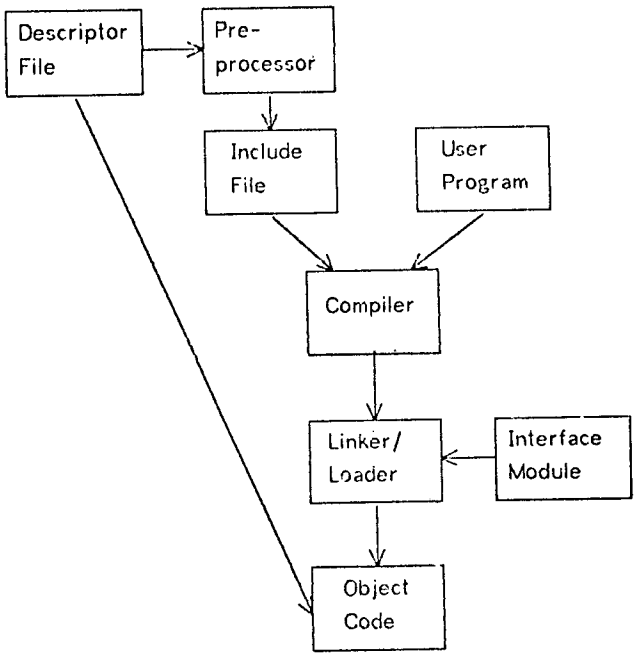

Figure 3: Using the Preprocessor

when processing a particular frame.

The preprocessor has two tasks: to verify the format of the descriptor file and to generate an *Include file* which is compiled with the application program. By using the preprocessor to check the descriptor file for conformity with the format expected by the interface module, the module does not have to do any error checking. This reduces both the time it takes to display a frame and the size of the module's code, and allows the preprocessor to do extensive checks which are infeasible for the interface module to perform. The second task of the preprocessor is to extract information from the descriptor file which is too time-consuming for the interface module to process and to put this information in an include file. The include file is a list of Pascal CONSTant declarations, one for each frame and for each command defined in the descriptor file (the same command can be used in different menus). Note that if the descriptor file is changed, a new include file must be generated by the preprocessor, and the application program must be recompiled. The identifiers declared in the include file allow the application program to refer to the frames and commands in the description file symbolically. Each command is equated to a unique integer value by the preprocessor. Commands which are not

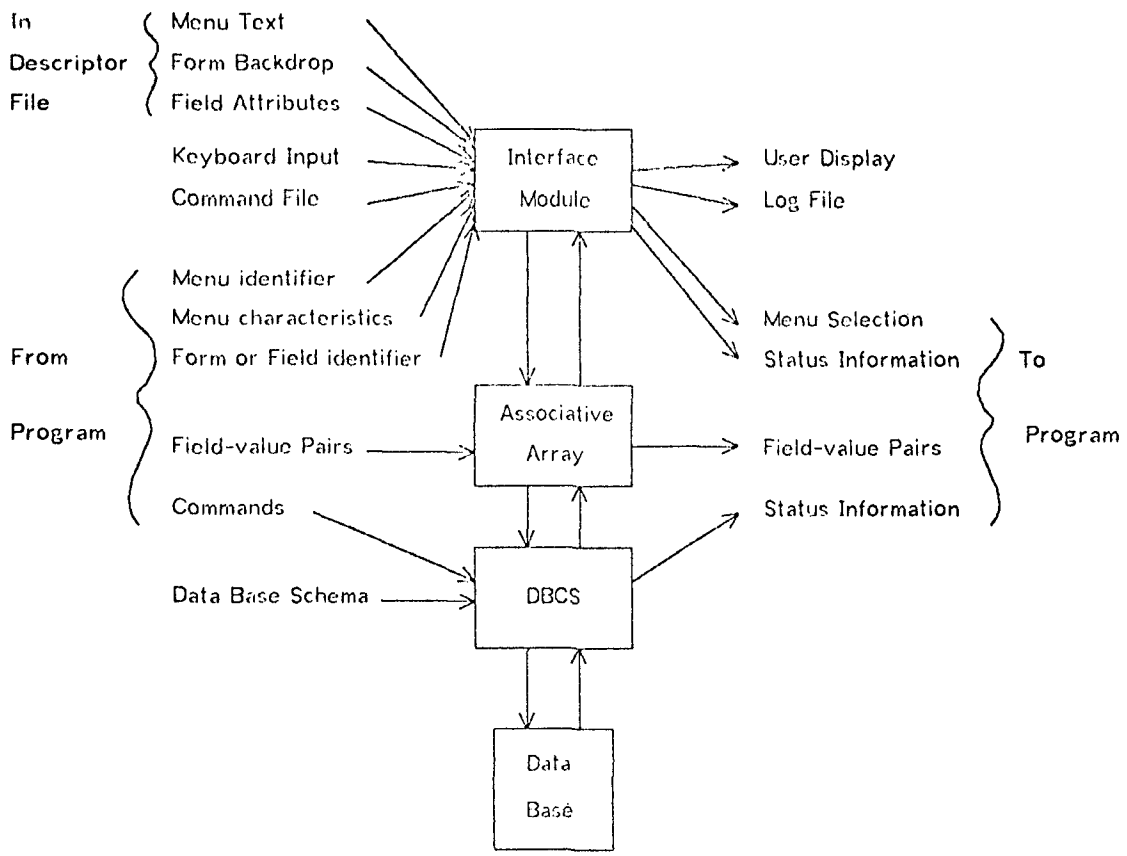

Figure 4: The User Interface During Execution

themselves valid Pascal identifiers must have an associated identifier specified in the description file. Each menu name is equated with a string (called the *menu descriptor*) that contains (in a packed format) the character position (i.e., the byte offset) of the start of the abbreviated and standard version of the menu in the descriptor file[1], the unique portion of each command that can be selected in this menu, and the integer value of this command (since the value is assigned by the preprocessor and does not appear in the descriptor file). The menu descriptor does not contain the character position of the detailed version since this version occurs immediately after the standard version, and can only be displayed by giving the *help* command while in the standard version. For the Customer List example given earlier, the menu descriptor would contain eight characters for the position of the menu (four each for the abbreviated and standard versions), twelve characters for the commands (four commands, with a one character name, a one character value, and a delimiting character) plus a final space (to signal the end of the command list) for a total of 21 characters. Each form name is equated with a string (called the *form descriptor*) which contains the character positions of the start of the backdrop and the start of the field attributes (eight characters total). Thus, the amount of main memory space occupied by the menu and form descriptors is minimal.

The organization of the program during execution is shown in Figure 4. An *associative array* (an array of strings indexed by strings) is used to pass data between the interface module and the *data base control system* (DBCS). After the user has entered data into a field of a form, the interface module stores the value of the field (without the noise characters and trailing blanks) in the associative array, indexed by the field name, which is one of the attributes of the field. To display a field, the value is retrieved from the associative array and displayed using the picture attribute. Thus the noise characters always appear in the field, but are not stored in the data base.

The data base management system is based on the CODASYL 1978 proposal [6]. The schema associated with the data base used by the application program describes the records in the

[1]Each byte offset requires 4 characters, enabling the descriptor file to grow as large as the UCSD file system allows.

data base and the fields that make up each record. To store a record in the data base, the DBCS retrieves the appropriate fields from the associative array, constructs the record, and places it in the data base. The reverse sequence of operations occurs when a record is retrieved from the data base. All conversions between data types are done by the DBCS, since the values of the fields are stored as strings in the associative array. In particular, numeric values are stored as strings. For variable length strings, the picture determines the maximum length and the string that is returned to the DBCS from the associative array is of the actual length (since trailing blanks have been removed). This aspect of the design makes the interface module as general as possible, and gives the DBCS complete control over all data representation issues. It is interesting to note that this design could also be successfully employed with a relational or hierarchical data base management system.

The interface module supports a small number of operations that can be applied to a frame. The INITIALIZE operation is used to specify the descriptor file, the initial verbosity (abbreviated or standard), and to prepare the module for the other operations. INITIALIZE sets the input device to the user's keyboard, and the log file to no file (indicating no logging). The CHANGEINPUT and CHANGELOG operations allow these two files to be set by the application program. Since the input device is not constrained to be the keyboard, it is possible to run the system from a command file, with the rest of the system behaving as if the input was coming directly from the user. By specifying a log file, a script of the user's input may be recorded. If the database becomes corrupted, it can be restored by using this file as a command file operating on a backup version of the data base. The DISPLAYMENU operation is passed a menu descriptor and returns the selected command as an integer. DISPLAYMENU handles all the details concerning the various menu levels and the generic commands. The returned command is guaranteed to be one of the commands that appears in the menu descriptor. The DISPLAYFORM operation is passed a form descriptor and displays the corresponding form on the screen. DISPLAYFORM also creates a field descriptor record whenever a field is encountered; this record, which contains the length and screen position of the field, is completely internal to the module[2]. The INPUTFIELD operation is passed a field designator (an integer), and returns a boolean status indicator (the entered data is placed in the associative array, as discussed earlier). The attribute line in the descriptor file that pertains to this field is retrieved, and the data entry proceeds. Analogous OUTPUTFIELD and MODIFYFIELD operations also exist, as do operations which, among other things, can input or output entire forms.

All the routines in the interface module are designed to optimize disk accesses, which are the bottleneck in floppy-based systems. A maximum of one disk read is required to initiate a frame display. When performing data entry on an entire form, several disk reads may be necessary, but these are not noticable since they occur between fields. Enough state information is retained between calls to the interface module to eliminate unnecessary reads from the disk. With the advent of Winchester technology disks, accesses to secondary storage is less of a problem. Using this new technology, an abbreviated menu would take less than 0.2 seconds to display, and a standard menu less than 2 seconds. At this point, the processing rate becomes a significant factor, especially when displaying forms containing many fields.

8. Acknowledgements

I would like to thank Norman Brucks for making this research possible by providing a suitable environment in which to build the system described in this paper, Merrie Brucks for helpful discussions on the psychological basis of user interface design, and several colleagues for comments on previous drafts.

[2]Since DISPLAYFORM deduces the screen position of the fields directly from the backdrop, there is no need for the implementor to explicitly include these positions in the descriptor file. Also, there are constructs that make the picture attribute independent of the length of the field (see the appendix), thus allowing the backdrop and the fie attributes to be changed relatively independently of each other.

References

1. S.J. Boise. "User behavior on an interactive computer system." *IBM System Journal 13*, 1 (1974), 2-18.

2. R.F. Brunt and D.E. Tuffs. "A User-Oriented Approach to Control Languages." *Software--Practice and Experience* 6 (1976), 93-108.

3. J.R. Carbonell, J.I. Elkind and R.S. Nickerson. "On the psychological importance of time in a time-sharing system." *Human Factors 10* (1968), 135-142.

4. R.G.G. Cattell. An Entity-Based Database Interface. Tech. Rept. CSL-79-9, Xerox PARC, August, 1979.

5. CODASYL COBOL Committee. *Journal of Development*. Secretariat of the Canadian Government, EDP Standards Committee, 1978.

6. CODASYL Data Description Language Committee. *Journal of Development*. Secretariat of the Canadian Government, EDP Standards Committee, 1978.

7. M. Genesereth. *An Automated User Consultant for MACSYMA*. Ph.D. Th., Harvard University, 1978.

8. Godden, D.R. and A.D. Baddeley. "Context Dependency for Recall Context-Dependent Memory in Two Natural Environments: On Land and Underwater." *British Journal of Psychology 66* (1975), 325-332.

9. J. Hurst and K. Walker (eds). *The Problem-Oriented System*. MEDCOM Press, New York, 1972.

10. T.C.S. Kennedy. "The Design of Interactive Procedures for Man-Machine Communication." *International Journal of Man-Machine Studies 6* (1974), 309-334.

11. B.W. Lampson and R.F. Sproull. An open operating system for a single-user machine. Proceedings of the Seventh Symposium on Operating System Principles, ACM, December, 1979, pp. 98-105.

12. K. Lantz and R. Rashid. VTMS: A Virtual Terminal Management System for RIG. Proceedings of the Seventh Symposium on Operating Systems Principles, ACM, December, 1979, pp. 86-97.

13. J. Martin. *Design of Man-Computer Dialogues*. Prentice-Hall, New Jersey, 1973.

14. J.M. McCrossin, R.P. O'Hara and L.R. Koster. "A time-sharing display terminal session manager." *IBM System Journal 17*, 3 (1978), 260-275.

15. A. Newell. Notes for a Model of Human Performance in ZOG. Carnegie-Mellon University Computer Science Department, August, 1977.

16. J. Palme. "A human-computer interface for noncomputer specialists." *Software--Practice and Experience* 9, 9 (September 1979), 741-748.

17. E. Rich. *Building and Exploiting User Models*. Ph.D. Th., Carnegie-Mellon University, April 1979.

18. G. Robertson, D. McCracken and A. Newell. The ZOG Approach to Man-Machine Communication. Tech. Rept. CS-TR-79-148, Carnegie-Mellon University Computer Science Department, October, 1979.

19. K.A. Shillington and G.M. Ackland (eds). *UCSD PASCAL Version I.5*. Institute for Information Systems, University of California, San Diego, 1978.

20. R.T. Snodgrass. An Object-Oriented Command Language. Carnegie-Mellon University Computer Science Department, 1980.

21. S. Treu. "Interactive Command Language Design Based on Required Mental Work." *International Journal of Man-Machine Studies 7* (1975), 135-149.

22. IFIP. *Proceedings of the IFIP Working Conference on Command Languages*, Lund, Sweden, August, 1974. Published as Command Languages, American Elsevier Publishing Company, New York, 1975.

23. P.L. Walton, R.R. Holland and L.I. Wolf. "Medical Guidance and PROMIS." *Computer 12*, 11 (November 1979), 19-27.

Appendix

This example illustrates the various components of a typical system. Comments concerning the components are in italics. A description of the format and use of these files may be found in section 7.

Descriptor File:

```
Example System                                              system name
?menu                                          the first frame in this example is a menu
Customer List                                                menu name
CustList                                                     menu identifier
append                                                       commands
edit
search
remove
?                                                  line separator, abbreviated menu follows
Customer List: A(ppend) E(dit) R(emove) S(earch) H(elp) Q(uit) ->
?                                                            standard menu follows
A(ppend)        This command allows you to add new customers to the Customer
                List.

                ...

Q(uit)          This command returns you to the main menu for the system.
?                                                            detailed menu follows
This menu deals with the Customer List, which is ...
?                                                  next complete screen of information
...
?form                                                        a form is next
CustForm                                           form identifier, followed by backdrop
Customer Name (First): @@@@@@@@@@@@@@@@@@@@
Customer Name (Last):  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Address:               @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
City [Palo Alto]:      @@@@@@@@@@@@@@@@@@@@
State [CA]:  @@    Zip Code [94306]: @@@@@
Telephone Number [(415) 858-1234 Ext. 5678]: @@@@@@@@@@@@@@@@@@@@@@@@@
...
?                                                            attributes follow[3]
"A(*)", " ", CustFirstName, If not known, press ENTER
"A(*)",,CustLastName,
"X(*)",,CustAddress,
"X(*)","Palo Alto",CustCity,
"XX","CA",CustState,
"99999","94306",CustZip,
"(999) 999-9999 Ext. 9999","41585812345678",CustTelephone,
...
?menu                                                        next menu
...
```

[3]The attributes are in the order of picture, default, field name, and explanatory message. "A(*)" expands to "AAAA...", with a length equal to the length of the associated field. This construct allows the picture attribute to be independent of the field size as indicated in the backdrop. An empty default will allow the value to be the empty string; a nonexistent default will not. Defaults can be provided by the main program by specifying a field name for the default rather than a string, causing the interface module to retrieve the default from the associative array.

Include File:

```
CUSTLIST='=;\f=;j]A 1E 2R 4S 3 ';                    the menu descriptor
CUSTFORM='=;bD=;ep';                                 the form descriptor
...                                                  other forms and menus
QUIT=0;
APPEND=1;
EDIT=2;
SEARCH=3;
REMOVE=4;
...                                                  other commands
```

Application Program:

```
PROGRAM Example;
USES InterfaceModule;                                declares procedures
...
CONST
    INCLUDE 'include file name'                      the include file
...
VAR
...
BEGIN
    InitMenu('descriptor file name', Standard);
    ...
    CASE DisplayMenu(CustList) OF
            Append: ...;
            Edit: BEGIN
                    ...
                    DisplayForm(CustForm);
                    ...
                    END;
            Search: ...;
            Remove: ...;
            Quit: ...
            END;
    ...
END.
```