OBLIQUE STEPS TOWARD THE HUMAN-FACTORS ENGINEERING OF INTERACTIVE COMPUTER SYSTEMS*

Raymond S. Nickerson
and
Richard W. Pew

20 July 1971

*Sponsored by the Advanced Research Projects Agency, Department of Defense, under Air Force Office of Scientific Research Contract F44620-71-C-0065, ARPA Order No. 890, Amendment No. 6, Code 1D20.

The authors are grateful to Mario Grignetti for his helpful comments on a draft of this report.

[redacted]

1

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 82_0510	2. GOVT ACCESSION NO. A951811	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) OBLIQUE STEPS TOWARD THE HUMAN-FACTORS ENGINEERING OF INTERACTIVE COMPUTER SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED INTERIM
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) RAYMOND S. NICKERSON RICHARD W. PEW		8. CONTRACT OR GRANT NUMBER(s) F44620-71-C-0065
9. PERFORMING ORGANIZATION NAME AND ADDRESS BOLT BERANEK AND NEWMAN CAMBRIDGE, MA.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 1993/04
11. CONTROLLING OFFICE NAME AND ADDRESS AFOSR/NL BUILDING 410 BOLLING AFB, DC 20332		12. REPORT DATE JULY 1971
		13. NUMBER OF PAGES 35
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

[illegible]

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents a potpourri of human-factors considerations pertaining to the design of general-purpose, interactive computer systems that are meant to be used by nonprogrammers. The reader is warned that it is informal, discursive and opinionated. The intent is to identify some specific problems, to offer tentative solutions to a few of them, and, most importantly, to stimulate more thinking on the part of both system designers and human-factors specialists along these lines.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

Abstract

This paper presents a potpourri of human-factors considerations pertaining to the design of general-purpose, interactive computer systems that are meant to be used by nonprogrammers. The reader is warned that it is informal, discursive and opinionated. The intent is to identify some specific problems, to offer tentative solutions to a few of them, and, most importantly, to stimulate more thinking on the part of both system designers and human-factors specialists along these lines.

The utility of an on-line, interactive, computational facility that is to be used by nonprogrammers will depend on (1) what capabilities the system provides, and (2) how accessible they are to the user. A scientist, for example, is interested in getting on with his research and is not likely to be enthusiastic about investing much time and effort in acquiring skills that do not have an obvious payoff in terms of his own research goals. There is nothing to be gained by providing him with a sophisticated system that will do many impressive things, none of which he is particularly interested in having done. Nor is there any advantage in giving him a system that will do some of the things he would like it to do, but is prohibitively difficult to use. But what are the characteristics and capabilities that a general purpose, on-line interactive facility should have? And how does one go about implementing them in any particular functional system?

The second of these questions clearly is a technical one, or, more accurately, it spawns a host of problems which must be answered in terms of programming or engineering techniques. The first question, however, is one of human needs and preferences. This being so, it might appear that the answer would be most readily obtained by asking the prospective user what he needs or wants. We think it is not likely to be as simple as that. A realistic appreciation of the features that an interactive system should have is most likely to be obtained as a result of first-hand experience with working systems.

The remarks in this paper are indeed based largely on first-hand experience with a small number of existing interactive systems and a second-hand (reading) acquaintance with a few others. The treatment of the subject is discursive and informal. No attempt has been made to formalize a set of design criteria or even to map an approach that might be taken to do so. Moreover, we make no claim to exhaustiveness in our enumeration of design considerations. Our intent is simply to identify what appear to us to be *some* of the features that an interactive system should have if it is to be generally useful to individuals whose main areas of interest lie outside the domain of computer technology itself. Many of the design features recommended below are incorporated in one or more existing systems; although, to our knowledge, no single system incorporates them all. Some of the features that will be noted will appear so obviously desirable as to preclude the necessity of even being mentioned. However, that it is painfully easy to overlook what is obvious to hindsight is attested by the fact that operational systems exist in which some of the most clearly desirable features are missing.

It will be evident that we focus primarily on general-purpose, scientifically-oriented—and, in particular, JOSS-like—systems (Baker, 1966). We hope, however, that the reader who is more concerned with special-purpose, problem-oriented, systems—reservation systems, cost-control systems, medical systems, instructional systems—will find some of the discussion germane to his area of interest. The need for effective user-oriented design is especially great in such special-purpose systems, inasmuch as the user is apt to see himself as even further removed from programming and other computer-related activities than is the user of a general-purpose system.

The recommendations that are made constitute a very "mixed bag." They involve various aspects of interactive systems—

languages, facilities, services, dynamics. (We have not paid much attention to the design of user terminals, a topic which is perhaps closer to conventional human engineering than are those which we do discuss. For discussions of some of the human-factors problems encountered in the design of keyboard terminals see Baker, 1967 and Dolotta, 1970. A more comprehensive discussion of human-factors considerations as they pertain to computer input and output devices is contained in Shackel and Shipley, 1970.) We have made no attempt to categorize our recommendations in any way, feeling that to do so would take us beyond the limited objectives of this paper, and perhaps create the impression of a more systematic treatment of the subject than is intended. The recommendations vary greatly in scope and specificity: general design principles are thrown in with "little tricks for making life easier for the user." They are offered quite frankly as opinions, and no effort is made to justify them with experimental data, or otherwise. If they stimulate further thought along these lines, or even the expression of opposing views, they will have served a useful function.

The Cardinal Assumption of the Uninformed User

Efficient interaction with the system should not be dependent on a knowledge of either the internal structure or the details of operation of either the system or the service programs. The user should be free to do his thinking at the level of the language with which he and the computer converse. There should be no need for him to be concerned with the way in which his program is represented within the machine, unless of course it is imperative to him that his program run at maximum efficiency, which usually will not be the case.

Training Requirements and Self-Teaching Capabilities

The system should require very little off-line training or instruction of the user. Ideally, it should be designed so that a novice can use it, at least haltingly, after spending a few minutes with a tutor or a manual, and can expect to learn to use it efficiently from the feedback provided by the system itself. Insofar as possible, the system should be designed in such a way that the most efficient and powerful approaches to problems are readily discovered by the user in the process of interacting with it. That is to say, the system should have a built-in teaching capability designed to facilitate the acquisition of that knowledge and those skills that qualify a user as an expert.

For example, it would be helpful to the novice user to be able to request the computer to give him examples of types of statements whose format he has forgotten, or not yet learned. To illustrate: a beginner might realize that the language allows "if" statements, but may not be able to put into an appropriate format a particular conditional that he wishes to write. He would then like to be able to put the system into a "teach" mode and ask it to give him some illustrative "if" statements—perhaps by simply typing "TEACH IF." The computer could thereupon produce a sequence of "if" statements in an order of increasing complexity until it had either satisfied the user or exhausted its supply of examples. Such a feature would also serve the more experienced user, who from time to time needs to refresh his memory regarding allowable statement formats.

A common practice is to build format information into the error diagnostics. For example, a format error might elicit a remark from the computer such as "The correct format is:" followed by an example of a correctly formatted statement representative of the type that the diagnostic program thinks the user was attempting to write. The objection to this procedure is that, if an experienced user is at the console, the lengthy output may be not only unnecessary but even bothersome. He may know exactly what his error is the moment it is pointed out to him that an error has been made. It would be in keeping with the policy of eliminating noninformative computer-to-user messages (see below) to provide the user with illustrative statements and detailed error diagnostics only in response to an explicit request.

Prompting can be another useful teaching technique and memory aid. To log in to the TENEX system,* for example, the user must type, in order and with appropriate terminators, the word "LOGIN," his name, a "password" and a job number (the latter for billing purposes). The experienced user does this more or less automatically; however, the novice or infrequent user can easily violate the format requirements, enter items in the wrong order, or forget to enter an item altogether. TENEX facilitates entry by identifying each of the components of the login procedure (except the first). The user need remember simply to type "LOGIN," followed by a special terminating symbol (the "escape" key on the teletype in this case). The computer will

*TENEX is a time-sharing system implemented on a DEC PDP-10 computer at Bolt Beranek and Newman Inc. Several of our examples are drawn from this system, in part because we happen to be familiar with it and in part because considerable attention was given to human factors problems by its designers. For descriptions of the system, see Myer and Barnaby (1971) and Burchfiel and Leavitt (1971).

then type "(USER)" and wait for the user to type his name, whereupon it will type "(PASSWORD)", and so on. The experienced user can suppress this prompting simply by using a different terminating symbol.

Updating Information

The need to train the neophyte is one requirement that occurs to everyone. A less obvious training requirement concerns the continuing education of the experienced but sporadic user. Few interactive systems are static. New procedures and upgraded versions of old procedures appear regularly. The chronic user who is on the system much of the time will assimilate changes gradually as they occur. The infrequent user will find it much more difficult to accommodate to changes that have occurred during a period of a few weeks or months that he has not used the system.

Typically this kind of training is provided by announcements made at sign-on time for two or three days following a change, and a memo to users may be issued to be read at their convenience. A better procedure would be to provide communication about system modifications contingent on their need. If a new format or command is defined that replaces an old one, the user should be trapped to a brief description of the new one and how to use it whenever he attempts to execute the old one. This procedure is rather like that used to correct for the dialing of an out-of-date phone number: the operator interrupts and provides the new number. When new procedures are introduced that supplement rather than replace others, use of the basic command should call forth a description of the supplemental procedure prior to execution of the command for the first three or four times the user applies it. The important point is that the critical dimension relating to the need for prompting the user's memory is not the time since the system change was made but the number of times that particular user has already been reminded of that change, and perhaps the recency of the last reminder. Such a procedure implies a bookkeeping burden for the executive program, but one that could be easily managed in a good system.

One simple expedient for getting updating information to users who need it, without forcing it on those who do not, would be to have the computer type the date (or perhaps the number) of the last change in the system, whenever anyone logs in. If the user is already aware of the change, he will simply proceed with the work session; if not, he can ask for a report. Following the typing of the report the computer would then give the date of the next-to-last change, and again, the user can decide whether he needs, or wants, to know about it. And so on.

Computer-to-User Messages

Computer-to-user messages should be designed to accommodate users representing all degrees of familiarity with the system. There are two types of computer-to-user messages that may occur in an interactive session: (a) those which the user intentionally elicits, either by requesting some specific outputs (program listings, values of variables, etc), or by inserting messages of his own composition into the body of his program, and (b) those that are preprogrammed into the basic system. We shall be concerned here only with the latter.

The purpose of such messages is to convey to the user some information that will facilitate his further progress with his program. Most commonly, they take the form of requests for specific inputs, of information concerning the state of the system, or of error diagnostics. In the latter case, an indication that an error has been made may or may not be accompanied by some information concerning the probable nature of the error. The problem is that of designing a message set and rules for message generation that satisfy the needs of users who represent every possible level of expertness in their interaction with the system. Novices will require lengthy messages which are completely self-explanatory; experts will prefer coded outputs which are as brief as they can possibly be made. Ideally, for the novice, every message should be meaningful the first time it is encountered. Satisfying this desideratum is in keeping with the objective of minimizing the amount of training a beginner must have before interacting directly with the system. It means, however, that messages should be written in a natural language (e.g., English) in whatever detail and with whatever degree of redundancy are necessary to ensure that they will be readily understood. Detail and redundancies that are helpful to a user who is learning the system will become sources of irritation, however, as he acquires skill. (One of the most reliable marks of the experienced user of an on-line system is his tendency to be exasperated by any delays which he perceives to be unnecessary. Given the opportunity, he would invariably replace lengthy messages with the briefest possible codes.) Even for experienced users, however, it is imperative that the computer do *something* whenever it receives a command that it cannot interpret. This is essential if one is to avoid the situation in which the computer is waiting for the user to input something interpretable, while the user is waiting for the computer to operate on what he assumes was an interpretable input.

Several possibilities suggest themselves for coping with the problem of conflicting desiderata of novices and experts concerning the form and content of computer-to-user messages.

1. Two separate programs. One possibility is to keep on hand two entirely independent systems which differ primarily, or only, with respect to the computer-to-user messages they generate. In one case, the messages, being complete and, hopefully, self-explanatory, are designed for the novice, the occasional user, and the visiting observer. In the other case, the messages are greatly abbreviated and intelligible only to the programmer or the user who has had considerable experience with the system.

2. One program, two message sets. It is, of course, oversimplifying things considerably to recognize only two types of users: novices and experts. It is more realistic to recognize that users represent a full spectrum of expertness. Any particular user masters a system only slowly over a long period of time. Moreover, different users, because of their own particular needs, may acquire skill with some aspects of a system while remaining relatively unskilled with respect to others. It may be advantageous, then, to allow the user himself to decide when he wishes to be treated as a novice, and when he wishes to attempt to play the expert. A simple way to provide this option is to include two complete message sets in the system, and to allow the user to switch at will between one and the other. Presumably, given such an option, the amount of time the user spends in the novice mode will decrease fairly regularly as he gains experience with the system.

3. "Yeah, yeah" signal. A third possibility is to provide the user with the means of cutting short a computer-to-user message while it is being typed out. For this approach to be effective, the user should be able to terminate any message, by pressing a single key, at any time during the message typeout. With this capability, the user need attend to the typeout only so long as it is informative. How much of a message he will want to see will depend, of course, on his familiarity with the system. Presumably, one's use of the interrupt option will become more frequent and more rapid as his experience with the system increases.

4. Two-part messages. A fourth possible approach is to (a) store each computer-to-user message in two forms—a concise mnemonic code and a complete self-explanatory statement, (b) *always* output the coded form of the message first, and (c) output the self-explanatory statement only if the user requests it, say, by responding to the coded form with "?". The advantages of this approach are several. First, the same program and the same mode of operation are appropriate for all users. Second, although decoded messages are always available when desired, the user never receives a lengthy message unless he specifically requests it. Third, the procedure facilitates the acquisition of just that knowledge which will make time-consuming messages unnecessary.

A combination of (4) and (3) would provide a particularly accommodating facility.

String Recognition

The capability for the computer to perform recognition on a partially complete character string effectively combines the principles of concise computer-to-user messages, prompting, and efficient training procedures. The string recognition procedure that is implemented in the TENEX system works in the following way. Whenever the user thinks that he has typed enough of a command string or file designator so that the intended command or file is uniquely specified, he may terminate the partially completed string with one of several terminators. With one terminator the computer either completes the typing of the designated string and waits for the next entry or parameter, or, if it cannot identify uniquely the string that has been terminated prematurely, it rings the terminal bell and awaits further input to complete the string. In a second termination mode the system accepts the abbreviation as it stands and either executes the command directly, or, if it cannot recognize the command or make a unique selection, it prints a "?" and aborts. In an earlier version of this recognition feature the computer took over for the user as soon as it had received sufficient characters and completed the string automatically. Given this procedure the user finds it easy to type accidently more than the requisite number of characters before the computer has time to take control. The result may be the typing of a few stray characters at the end of the command that at best are misleading and at worst confound the beginning of the next input. The string-recognition feature, as currently implemented in TENEX, is especially convenient if it can be applied to terms defined by the user himself as well as to system-defined commands.

Default Values and Conditions

Often in interperson conversations, information is exchanged by default. If one mentions Paris, for example, it is likely to be assumed that he is referring to Paris, France; had he meant Paris, Maine, he would have been expected to say Paris, *Maine*. Similarly, in the case of man-computer interaction it is sometimes possible to assume what unstated values of program parameters should be, and to assign them by default whenever the user does not explicitly indicate otherwise. Default conditions make it possible to build into the system considerable sophistication that can be exploited by the user as far as he wishes, or to the degree consistent with his level of training. As an example consider the file designation procedure used by the TENEX system. A complete file designator consists of five parts, and might look as follows:

```
ALPHA. F4; 3; A12345; P77520̸2
```

Part I (ALPHA in our example) is the file name assigned by the user. The system will recognize an abbreviation (first few letters) of the name so long as no other file name would be abbreviated the same way. Part II (F4) is the file extension, which tells the system what kind of file is involved. It is also subject to the automatic recognition procedure. Part III (3) is the version number. When creating a new file the default value of the version number is one. When creating a new version of an old file the default value is one greater than the last number used with that file name and extension. When deleting a file the earliest version number is assumed unless the user explicitly specifies a higher one. Part IV (A12345) is the account number to which page charges will be assigned. If the user defaults this number, the account to which his compute time is charged is assumed. Part V (P7752Ø2) describes a protection or privacy status for the file. If no number is specified it is assumed that any other user may read the file but only the creator of the file may write into it or delete it. Note that for a typical user Parts I, II and occasionally Part III are sufficient to declare most files and it is the exception that requires further specification.

In some cases in which it is not clear in advance what the best default value is, it might be appropriate to sample user opinion or to collect statistics on the most frequently used value in order to determine what it should be. When it is important for the user to know exactly what he defaulted, the machine should prompt him with the defaulted value. It is important, for example, for the TENEX user to know his extension and version number, but the account and protection information are not displayed unless specifically requested.

Program Component Identification

There should be a straightforward way of structuring a program and of identifying its components. Perhaps the most common structure in conventional programming is that of a heirarchy: programs, subprograms, routines, subroutines, etc. There is every reason to expect that this will be equally true of interactive programming; hence, there is need for a means of identifying program components in such a way as to make it possible to refer to any level in a hierarchy of arbitrary depth.

Several of the current JOSS-like systems provide for a two-level organization of a program in "parts" and "steps." The convention is to identify steps with decimal numbers, the integer part of the number designating the part to which the step belongs. Reference can then be made to, and operations performed upon, either individual steps or parts as wholes. Thus, for example, the command "DELETE PART 3" would, in effect, delete steps 3.1, 3.12, 3.2 and any other steps identified with a number whose integer part is 3. The restriction of two levels imposed by this scheme might not be a serious limitation for the casual user of a system; however, it probably does represent an unnecessary constraint for the more experienced user. Moreover, it is a limitation that is removed by simply making the convention that when a command can appropriately reference more than a single step (e.g., DELETE, TYPE, DO), the command will be understood to refer to all steps whose most significant digits correspond to the number in the command statement. Hence, the command "TYPE PART .1324" would cause the typing of steps .13241, .13242, .132431, and any other step whose number began with .1324. If the user wished to refer to a single step, he would, of course, have to use enough digits to identify that step uniquely. For example, assuming that his program contained each of the above step numbers, in order to have the single step .1324 typed, he would have to say "TYPE .1324Ø."

List-processing languages such as IPL and LISP are not organized in terms of numbered steps, so this convention does not apply. In LISP, program components are "symbolic expressions," each of which is comprised of a function and its arguments. The arguments of a function may be functions in turn, so that these programs also have a hierarchical structure. Expressions or subexpressions may be identified via the appropriate function names. List-processing languages are less likely to be of concern to the nonprogrammer computer user than are the JOSS-like languages—at least in the near future—so they are given little attention here.

Editing Capabilities

The system should provide flexible editing and error-correcting capabilities. It is convenient to make a distinction between two broad classes of editing and error-correcting operations: those which may be performed on a program component or step as it is being composed, or *local* operations, and those which may be performed on steps which have already been inserted into the program, or *remote* operations.

There are two local operations which, from the user's point of view, are needed: one to delete the last character typed, and one to delete the entire step or program component currently being entered. Each of these should be executed by striking a single-control character. The operation deleting the last character should be iterative, allowing the user to delete the last n characters typed. In the case of teletype or typewriter input it should not be possible, with this operation, to delete elements past the first character of the current line or program component because it becomes very difficult to keep track of exactly what was deleted. This restriction is not important in the case of a CRT terminal where the consequences of deletion can be portrayed literally to the user; i.e., the deleted characters actually can be made to disappear and new ones to appear in their places.

When text is being displayed on a CRT as it is being typed, a cursor or underscore should be used to show the location of the next character to be typed. This is especially helpful when nonprinting characters (spaces, tabs, carriage returns) are being used in formatting tables, labeling graph axes, etc.). A further convenience to the user would be an alternate mode of display in which nonprinting characters are explicitly represented by special symbols.

A flashing cursor can be helpful when backspacing over displayed characters for erasure or editing. Rule: have the cursor flash whenever it is pointing to the location of a character that has just been deleted from memory. Again this would be particularly useful in the case of nonprinting characters.

There are four remote editing operations that are essential to an on-line system. They are the operations of deletion, replacement, insertion, and revision. The operand may be a variable, a step or other program component. Given a step-numbering scheme such as that described above, the remote operations of step deletion and insertion are self-evident. One advantage of such a scheme is that it obviates the renumbering following the deletion or addition of steps. For example, given a program comprised of steps .11, .12, .13, and .14, deletion of step .12 and insertion of two additional steps between .13 and .14 would not necessitate renumbering any of the original steps that are retained, even though their ordinal positions in the program have been changed. The steps of the program following the indicated changes might be numbered .11, .13, .131, .132, and .14. Step replacement would be accomplished by simply writing a new step and assigning it an old number, the system being designed so that whenever a step is given the same number as that of a previously entered step, the original step is replaced by the new one.

The delete operation can of course cause grief when supplied with an erroneous argument. An easy way to guard against this event is to force the user to think twice about any such command. In PROPHET (Castleman, *et al.*, 1970), a CRT-oriented chemical/biological information-handling system, the effect of a delete command is to have the to-be-deleted element blink on the display. The user then must verify that the blinking element is in fact the one that he wishes to delete.

A system that allows only the three remote operations of deletion, replacement, and insertion would be reasonably adequate for many applications; however, to be truly efficient, it should include, in addition, a capability for revising steps or other program components without completely retyping them. In many instances the user will want to change only those portions of a step that are in error, while retaining those portions that are correct. It is an inconvenience, for example, to have to retype a lengthy and involved algebraic statement to correct a single erroneous character. The need here is for deletion, replacement, and insertion operations which can be performed on *elements* within a step. The more sophisticated systems provide editing commands for searching program components for particular characters or character strings, and for performing delete, replace, or insert operations relative to the result of the search.

In addition to providing these component editing capabilities it is also important not to place artificial constraints on the ways in which they may be used. It should be permissible to intermix freely editing commands and to make up strings of commands to be executed as a unit. For example, to change N=N+1 to N=N+2, one might want to write an editing procedure that would search for the string N=N+, delete the next character in the line and insert 2 in its place. In the TENEX version of TECO, which is a language used primarily for the purpose of editing, this is accomplished by typing the string

```
SN=N+$DI2$$
```

where the S, D and I indicate search, delete and insert, respectively. The first and second dollar signs terminate the search and insertion strings, and the third executes the string of editing instructions.

A common practice in algebraic interactive languages is to reject an input string if the computer detects a syntactic error and to inform the user of why the input was unacceptable. We recommend instead that the aberrant string be retained in the buffer and the computer automatically shifted into an editing mode so that the user may choose to delete the entire string or, if possible, to correct it by changing one or two erroneous characters. It is more than mildly irritating to complete the typing of a complex algebraic expression only to find that it must be completely reentered in order to add one forgotten right parenthesis.

Direct and Indirect Commands

The system should allow both direct and indirect commands.

By direct command is meant a command that is to be executed immediately; an indirect command is one that is to comprise a component of a program, and that will be executed in the course of the execution of the program to which it belongs. The direct-command capability allows the computer to be used as a powerful desk calculator for such purposes as evaluating mathematical expressions, generating tables, and plotting functions on a one-shot basis. It also serves as an important tool for debugging and editing active programs. Indirect commands provide for the construction of programs. Virtually all conversational languages include both direct and indirect commands. In some cases, however, direct commands comprise a minimum set (DO, RUN, EXECUTE), in which case in order to use the computer as a desk calculator one must enter an indirect command and then execute it as a program.

Arbitrary Starting Point

The user should be able to start or restart his program at any point. In particular, after fixing an error that has caused a running program to halt, he should be able to restart the program at the point at which it stopped.

Variable Names

In composing programs, the user should be free to assign names to variables in a way most consistent with his own mnemonic conventions. Ideally, he should be allowed to call variables anything he wants; in practice, other considerations may place a limit on the number or types of characters a name may be allowed to contain. If a limit must be imposed, five or six characters per name would probably be adequate for most users; three characters per name is perhaps tolerable; a single character limitation (even with subscripting) is a definite handicap.

Language Modification and Abbreviations

A means should be provided for the user to modify the language and redefine terms. For example, an individual who finds himself using a small set of commands very frequently might find it economical to replace each of these commands with a single-character abbreviation. Insofar as possible, he should be allowed to establish equivalences of this sort.

One should also be able to define and use abbreviations for such things as variable names. For example, PROPHET, the chemical/biological information-handling program mentioned above, permits one to give a variable such a name as "MOLECULAR FORMULA OF ASPIRN," and then define and use an abbreviation such as "MA" (Castleman, *et al.*, 1970).

The user should not, of course, be allowed to make language changes that will affect other users in any way.

Address Arithmetic

Languages for which a step is the basic program component (e.g., JOSS-like languages) should permit the changing of step numbers for any specified program segment with a single command. For example, a command like "CHANGE STEPS .21 to .46" might be used to replace all the step numbers beginning with .21 to new numbers beginning with .46, leaving the less significant digits unchanged.

Algebraic Expressions as Inputs

The system should accept and correctly interpret any evaluatable algebraic expression in any case in which a number is an admissible input. As a simple but important example, one should be able to input fractions as *fractions*, that is, one should be able to insert 1/17 as opposed to .Ø58889. The importance of this capability does not stem from the fact that a fraction is easier to type than a decimal (although if one wants accuracy, he will, in general, have to type several more characters in the latter case), but rather from the fact that, if the user has the fraction to begin with, converting it to a decimal number involves a task that the computer, not he, should perform. The ability to input fractions directly is a particular advantage to the user who is dealing extensively with probabilities.

Identification of Precision Limits

The limitations of the system with respect to numerical precision should be explicit in the output. The system should not produce numbers with more significant digits than are justified by the computational accuracy of its number-handling procedures. For example, if the system can assure only ten bits of accuracy in its number representation, it should not output numbers with more than three significant (decimal) digits. Since most machines use floating-point arithmetic, which allows the manipulation of numbers whose magnitude is far beyond the precisional limits of the system, there must be some straightforward way to represent arbitrarily large numbers so that the accuracy limitation is obvious. One possibility is to express all numbers in scientific notation with the fractional part being limited to the number of digits implied by the precisional capabilities of the system. Another possibility is the use of filler symbols. For example, given a limitation of three decimal digits of accuracy, the number 365,741 might be represented as 366,xxx. It should not be represented as 366,ØØØ, since in this case the limitation is not obvious. The system should round the output to the least significant digit; it should not truncate. In short, when a user receives a number from the computer, he should be able to assume that it is exactly the number that he would have obtained had the computation been done by hand, and rounded off to the same number of significant digits.

Formatting Options

The system should provide formatting options specifically designed to assist the user in making his program easy to read. Extra spaces and carriage returns should be freely allowed and should be preserved in storage at the level of the symbolic program. In scientific programming, one frequently wishes to construct algebraic statements involving several depths of nested parentheses. Parenthesizing errors are very easy to make, and can be frustratingly difficult to find. It would be a help to have several, say three, different characters, e.g., (, [, {, for formatting algebraic statements. These characters could be equivalant as far as the program interpreter is concerned, but the distinction should be maintained at the level of the conversational program. Such a feature would facilitate the construction of complex algebraic statements and would simplify the process of finding errors when they occur. It would be particularly helpful if the different parenthesizing symbols were different sizes.

Another useful formatting convention, easily implemented with a typewriter as the I/O device is that of color-coding the dialogue, printing user-generated text in one color and computer-generated text in another (Baker, 1966).

Procedure Definition

There should be a straightforward means of defining and storing generalized program components and retrieving them for incorporation as elements in programs or higher-order components. Having once written a particular generalized program component (procedure, function, macro, subroutine), one should not have to write the same component again. Heavy users of an interactive system are likely to be developing many programs having common components. The prospect of developing a library of program components especially tailored to one's own needs is perhaps one of the most compelling enticements that a computer system can offer to a prospective user.

Procedure Library

The system should maintain a central public library of programs and procedures that are available to all users. The library should be designed to expand as users generate new programs of general interest. Every user should have read-only access to the library on a continuous basis. He should not, however, be able to enter programs directly into the library. One possible scheme for allowing a user to contribute to the library would be to have him deliver a program to a temporary file which is periodically examined by the system supervisor or librarian for the purpose of updating the library file.

Compilation Capability

A system designed specifically for scientific and engineering applications probably should have a compilation capability. The interpreter should be used for exploratory programming; however, when a program is to be used frequently for production runs it should be compiled. This is especially true when compilation results in noticeably shorter system response times. It is essential, however, that such a compiler accept as input the program as it was written for the interpreter.

File Storage

In cases where lengthy work sessions are anticipated, it should be possible for the user, when terminating a session with work unfinished, to leave the system in such a state that, upon reentering it at a later time, he will be able to resume his work exactly where he left off. This means providing the user with the capability to store his virtual core in a long-term storage medium such as magnetic tape or disc, and to retrieve it upon reentering the system. The user should also be able to maintain files of his own subroutines, programs and data sets.

Short Interruptions

In addition to the capability for the resumption of work after indefinite periods, there should be a simple procedure for allowing brief interruptions in a work session. It frequently happens in the course of an on-line session that the user finds it necessary or advantageous to leave the console temporarily (e.g., to attend to an unexpected visitor or telephone call, or to dispose of some pressing business—or perhaps to cogitate about his program or some results he has obtained from running it). If it is likely to be several minutes before he will return to the computer, and particularly if he is being charged on the basis of on-line time, he will want in such cases to be able to take "time out," to tell the computer it can forget about him until such time that he indicates that he is ready to resume the session. The procedure for effecting such a recess should be less involved than that used to store a system for reactivation in the indefinite future. It should not, for example, be necessary explicitly to create files on a long-term storage device. Ideally, to initiate the time out, the user should be required to do nothing more complicated than to press a special function key, or perhaps to type "time out" or "wait" or some such thing. Resumption of the session should be effected by an equally simple procedure.

Program and File Information

The system, on request, should be able to provide the user with information concerning the status or contents of his program. It should be able to produce, at the minimum, a copy of any specified segment of the user's program, a list of variables, functions, procedures, macros that the user has defined, a table of contents of the user's files or previously stored programs, values of variables, indexes, subscripts, etc.

Status and Control Information

The user should be provided continuously with status and control information. At the very least, he should be informed as to whether he is waiting for the machine or it is waiting for him. (The JOSS system provides this information via a red and a green light at the console that indicate whether the computer or the user is controlling the typewriter [Baker, 1966].) Given that the user is waiting for the computer, he might like to know: (1) is the computer currently working on his problem? (2) is it waiting for a peripheral device like a tape unit or line printer? (3) is it waiting in a queue for its "slice" of time? or (4) is the system dead?

Feedback to the user is particularly important when the length of the delay to be expected is unknown. For example, a long pause after some data have been entered can make the user wonder if he has entered data incorrectly, or possibly has not properly signaled the computer that he is done. The computer should signal receipt (or acceptance) of entry immediately, even though there may be a delay before the next entry can be accepted, or before there is a substantive response (Poole, 1966).

In some systems it is practical to include an auxiliary display at the terminal that provides the user with his current status with respect to these alternatives, but in systems operating over telephone lines this may not be economically practical. An alternative that seems to be quite effective is to provide a status command with which the user can interrupt the ongoing computation long enough to have printed a computer-to-user message describing both his current status (running, I/O wait, etc.) and give the cumulative log-on and CPU time used to date. The system is then restored immediately to its former status with no loss of priority. In the course of a long computation, user-initiated periodic status interrupts of this sort can provide quantitative information regarding how much of the machine's time one is getting per unit of elapsed time.

The system should be able to tell the user how much time he has used since the beginning of the session, or since some specified date. It should also be able to produce a statement of charges accrued since the beginning of the current billing period against the user's job number or account.

System Dynamics Information

If the system dynamics (e.g., response time) change significantly with the load, as they usually do, it would be a convenience to the user if he could get an indication of what the load is before deciding whether he should get on. At a minimum the system should be able to answer the question: How many users are now on line? Other, and more helpful, items of information are, in principle, obtainable (e.g., mean system response time to a request for a given time slice over the last n minutes), but only at a somewhat greater cost in overhead program execution.

Fail-Safe Provisions against Potentially Fatal Operations

Users make mistakes. They enter commands they did not intend and sometimes discover what they have done too late to avoid the dire consequences. If one deletes a program, or a file, by mistake, for example, in most systems there is no provision for recovering from such an error. The program, or file, is gone and would have to be reentered in its entirety. Provisions can be made, however, either for decreasing the probability of such errors or for facilitating recovery from them when they do occur.

A simple measure for decreasing the probability of such errors is to require for commands that modify stored programs or files (e.g., DELETE, KILL, MODIFY) some confirmation in addition to the usual command terminator. This is tantamount to forcing the user, after issuing a potentially destructive command, to indicate explicitly, "Yes, that is what I meant to say." Such a fail-safe measure is implemented in the PROPHET system, as noted under Editing Capabilities, above.

An alternative way of dealing with this problem is to implement procedures for recovering from the erroneous entry of potentially disastrous commands. Some systems, for example, have implemented "UNDELETE," "UNDO," or "RESTORE" commands. In BBN TENEX, UNDELETE restores the file to the user's directory after it has been inappropriately deleted. It may be used any time up until the user logs out of the system, at which time his deleted files are expunged. In TENEX-LISP, UNDO undoes the effects of a program execution. "UNDO PART 4" would restore the program to the status it had, complete with the variables and constants as they were, before part 4 was executed.

No Invisible Mistakes

Interactive systems make frequent use of nonprinting characters as control characters. It is important to the user who is attempting to diagnose an error that it not be possible to have an error hidden because it involves the application of a nonprinting character. This can be avoided by having a special character echoed at the terminal for every one that does occur in a character string.

Conditional Dump of Stacked Input

When a full duplex terminal is in use, in which the type bar is controlled by the computer and every typed character is echoed through the machine, it is possible to type at the keyboard while the computer is occupied with computation. The typescript that is entered this way is not reflected back to the terminal until the computer releases control of the interaction. If the computation is ended appropriately all is well, but if the computation is terminated prematurely because of an error or because of an unanticipated program branch, then the preentered typescript is appended to the end of the error message and is interpreted as the beginning of a new, but, in this case, inappropriate message. Whenever an error termination like this occurs, the system should automatically dump the prestored typescript and leave the user with a clean slate to deal with the error condition.

Report Quality Output

The system should be capable of producing output of a quality acceptable for incorporation in official reports. This goal is somewhat more easily realized with typewriters or with MODEL 37 teletypewriters than with MODEL 33 or 35 teletypewriters, since in the former cases one has a conventional character set, including both upper- and lower-case characters. There is, however, a considerable need for research into the problem of improving the design of keyboard devices that are to be used as computer terminals (see Dolotta, 1970). The identification of an adequate character set is only one of the many problems that arise in this context.

"Sense" Switches

Most computers provide the programmer with a set of toggle switches (usually referred to as "sense switches") on the console, each of whose positions (up or down) can be examined by the program. By making the course of the program at different points contingent on their positions, the programmer can make it possible to control the flow of his program at run time by manipulating the appropriate switches. Such real-time control of a running program could be a very great convenience to the user of an interactive system, and could be provided by means of a set of sense switches located at the remote terminal. A cutout overlay that accompanies the program to be run could be used to remind the user of the status and meaning of each sense switch, which could change, of course, as a function of the program being run.

User Interrupt

We may think of the user-computer interaction as always being under the control of either the user or the computer. Whenever it is the user's turn to "say" something, we say he is in control. He may actually be typing a user-to-computer message, or he may be scratching his head thinking about what to type; in either case, if the computer is waiting for an input from him, we say he is in control of the interaction. Similarly, the computer, while in control, may be outputting a computer-to-user message, or it may be executing a program in preparation for outputting a message. Normally, control passes either from the user to the computer, or vice versa, at the termination of a message. That is, one of the communicants regains control by virtue of the fact that the other relinquishes it, having completed a message, and having nothing more to say at the moment. To a large extent, it is this continual exchanging of control, the give-and-take dynamics of the situation, that justifies describing the interaction as "conversational." There is a need for one exception, however, to the normal way of passing control from the computer to the user: the user should have the ability to interrupt. That is, he should be able to seize control of the interaction at any time, without waiting for the computer to relinquish it.

The need for this capability is most clearly seen in the case of a lengthy computer output which, from its beginning, is obviously erroneous. Suppose, for example, that the user has programmed a loop to generate a lengthy table, and that by the time the first few values of the table have been typed, it is clear that there is something wrong with the algorithm. In such a case, the user should not be forced to wait until the entire table has been generated before regaining control of the interaction. He should be able, by pressing a single key, to cause the computer to stop what it is doing and to await further instructions from him.

Background Execution Option

The efficiency of an interactive system could be increased by providing the user with the option of "detaching" his program from interactive control at the terminal and having it run as a low-priority background process. Suppose, for example, a particular application involves developing a procedure for generating fairly lengthy tables. While developing and debugging the procedure, the user wants to be on-line. Once the procedure is operating satisfactorily, however, he may simply want to leave it alone and let it generate its output. In such a case, the user would like to be able to leave the terminal and return after the tables have been completed. Moreover, unless there is some urgency for an immediate result, he would probably be

content to have it generated at the computer's leisure, espec-
ially if background-processing time were charged out at a lower
rate than on-line time.

Programmed Logout

There should be an instruction to discontinue service that could be appended at the end of a program, thus permitting the user to log out of the system and disconnect the terminal in-directly. If one has written a program that will run for a considerable time without intervention, it should not be neces-sary for the user to stay around simply to pull the plug at the end of the session. As a fail-safe protective measure against program malfunction, it would be a convenience for the user to be able to specify a time at which his program should be automat-ically terminated in the event that it is still running.

Complaints and Suggestions

The system should have a complaint or suggestion input capability. Ideas for system improvement frequently occur to a user in the process of interacting with the system, and are forgotten by the end of the session. Similarly, a minor mal-function, unless it is serious enough to terminate the session, is apt not to be remembered. It would be a convenience to the user, and it should be an aid to the system managers, if it were possible to insert a complaint or suggestion directly into an appropriately designated file at the point during the on-line session when the occasion arises. A hard-copy record of the file could then be made periodically and might prove to be a valuable source of information when attempting to improve the system.

REFERENCES

Baker, C. L., 1966. JOSS: Introduction to a helpful assistant. *Memorandum RM-5058-PR*. The Rand Corp., Santa Monica, Calif.

Baker, C. L., 1967. JOSS: Console design. *Memorandum RM-5218-PR*. The Rand Corp., Santa Monica, Calif.

Burchfiel, J. D. and E. M. Leavitt, 1971. TENEX: User's Guide. Bolt Beranek and Newman Inc., Cambridge, Mass.

Castleman, P. A. *et al.*, 1970. THE PROPHET SYSTEM: A final report on Phase I of the design effort for the chemical/biological information-handling program. National Institutes of Health and Bolt Beranek and Newman Inc.

Dolotta, T. A., 1970. Functional specifications for typewriter-like time-sharing terminals. *Computing Surveys*, 2, 5-31.

Myer, T. H. and J. R. Barnaby., 1971. TENEX: Executive language manual for users. Bolt Beranek and Newman Inc., Cambridge, Mass.

Poole, H. H. *Fundamentals of Display Systems*. Spartan Books, MacMillan & Co., 1966.

Shackel, B. and P. Shipley. Man-computer interaction: A review of ergonomics literature and related research. EMI Electronics Ltd., Report No. DMP 3472, Feb. 1970.