A general purpose dialogue processor

by JAMES L. BLACK
Science Applications, Inc.*
San Francisco, California

ABSTRACT

This paper describes work done on the Automatic Program Generator project at the Sperry Research Center. The overall aim of this research is to discover techniques that will make computers more directly accessible to non-technical users.

The approach taken is based on the use of interactive dialogue, using CRT terminals or work stations. A general purpose Dialogue Processor has been implemented whose function is to facilitate the creation and management of interactive dialogues. A high-level Dialogue Specification Language (DSL) is used, in which dialogues are represented as choice trees. Interactions with a user translate into selection or rejection of available paths plus any required data entry. Trees may be specified which present steps in the formulation of commands or transactions in the way that appears most natural to the user. User responses are then mapped into an appropriate format for output to the process that is to be dialogue driven. A Dialogue Editor enables the user to backtrack during a dialogue session, or at a later date to reformulate his responses.

The Dialogue Processor is a universal facility which has many potential uses; essentially, it can front-end any parameter or transaction driven system. It is currently being used as an interactive program specification technique in conjunction with an application customizer. Among other possible applications are its use as an aid to formulating complex command language statements (e.g., JCL) and as a query language interface for data bases.

We believe that the approach described in this paper, coupled with declining hardware costs and other technological advances, makes it possible to extract increasing benefits from interactive dialogue, while minimizing some of the traditional drawbacks.

IMPORTANCE OF EASE OF USE

It would be hard to overstate the importance that ease of use will assume in future computer products. This feature is given prominence in the list of design priorities for all new Sperry Univac products; and indeed, it is being stressed increasingly by the computer industry as a whole.

Ease of use becomes critical to the success of small business systems aimed at the first-time user market. For such customers, the cost of programming, if it were carried out in the traditional way by hiring application programmers, would easily overshadow the rental cost for the machine. At the lowest level, small business system customers have typically depended on the vendor, or a third party, to supply application programs. These generalized application packages have then to be tailored to the particular customer’s business methods by the vendor’s support personnel, either manually or in a more or less automated fashion through an application customizer. Such a situation arose when Sperry Univac OED decided to introduce the BC7 small business computer. The Sperry Research Center participated in a joint effort with Sperry Univac to develop an interactive customizing facility for application packages, some details of which will be described later in this paper.

Ease of use considerations are not, of course, restricted to small machines; probably the most fertile area is to provide facilities for a whole range of users to gain access to distributed computer networks for a variety of purposes, with particular emphasis on shared data base applications.

STANDARD SOLUTIONS TO PROVIDING EASE OF USE

Easy to use languages

The traditional approach to providing solutions to the ease of use problem has been by designing languages. Because of the number of languages that have been created, and the lack of consensus on what are the criteria for deciding whether or not a language is easy to use, a discussion of computer languages will not be attempted here. While some general purpose languages make a claim to be easy to use, such as COBOL, BASIC or APL, another approach has been the design of special purpose languages. Very High Level Languages could be characterized as languages in which the user specifies “what” he wants done rather than “how” to solve the problem; such
an approach, almost by definition, limits the field of applicability of the language. Other examples of special purpose languages are data base "query" languages, such as Sperry Univac's UNIQUE and QLP languages which operate in IMS 90 and DMS 1100 environments respectively. These languages permit an end user to perform a restricted number of file operations, such as retrieval and/or update of data items which satisfy search criteria submitted by the user.

The two above mentioned, as well as many other languages, are designed to be used through terminals, in which environment the user both receives diagnostic and prompting messages at the time of submitting his program, and in many cases will receive output at the terminal on execution of the program. Interactive operation is one of the keys to ease of use.

Another approach to ease of use is to make the computer language approximate to natural language as closely as possible. Thus, there have been many efforts and proposals to use English as a programming language. We are seeing much effort in the Artificial Intelligence community being directed towards English-understanding systems. However, even if it were conclusive that natural English was well adapted to expressing user needs (which is arguable), we are still a long way from being able to propose practical and economic systems using English as a programming language.

**Other man-machine interfaces**

Procedural languages are one form of man-machine interface. Another way we frequently communicate with the machine is through a command language (e.g., JCL). However, there are also many non-linguistic methods of getting the computer to execute the functions we desire. The most basic of these is typified by the computer operator's use of a console consisting of switches and lights. In some cases a standard terminal may be used in console-like fashion.

An off-line analogue to setting switches is the use of forms. The use of a check mark or one to one of a set of code symbols in a fixed position on a form is the equivalent to the setting of a switch, once the form is transcribed to a machine readable medium.

A third non-linguistic interface is interactive dialogue. In this case the operator responds to machine requests for decisions or values which will determine the succeeding course of events.

In general, non-linguistic interfaces are much less versatile, offering only a restricted range of possibilities, when compared to a general purpose programming language. However, versatility is not always desirable or necessary when ease of use is the goal, and the amount of built-in prompting and control possible with non-linguistic systems makes them candidates for further development.

**EASE OF USE THROUGH DIALOGUE**

The use of computer dialogue is now, of course, commonplace; we see it used in a host of applications. It is worthwhile, however, to reexamine some of the pros and cons of this technique.

**Advantages**

The advantages of dialogue driven systems may be summarized as follows.

- Learning is reduced or eliminated. The dialogue serves as a prompting mechanism or, depending on its verbosity, even as a tutorial, to guide the user's next response.
- User responses can be minimal. In a typical case an entire line of information might be displayed to the user, and his reply could consist of a single keystroke: Y for yes or N for no.
- User response can be validated immediately. Any unacceptable responses can be detected at the ideal time and the question either simply repeated or a diagnostic message displayed.
- The sequence of computer initiated questions is dynamically changeable, depending on the response to previous steps; thus, an appropriate subset of questions can be asked, depending on the problem, without the user's being exposed to irrelevant dialogue.
- As the dialogue proceeds, the user can save and review the previous interactions.

It is clear that dialogue is an invaluable tool in making computers easier to use. Well designed dialogue is "natural," if the computer's questions are phrased in terminology that is clear to the user. We can consider the computer as playing the role of an "automated consultant," asking all the necessary questions, and only the necessary questions, for some particular purpose. The function of the dialogue may be to control a process directly in real-time (such as in many time-sharing systems dialogues), or to extract information from the user. If information gathering is the purpose, the answers obtained from the user may represent the specification for some subsequent process; i.e., the responses could consist of language statements or a set of parameters.

Where a complex set of specifications is required, a form, or set of forms, is often employed. An example is the specification of a report using RPG, in which various forms describe the format of the output, the nature of the input and any calculations, summarizations or editing that is required. An alternative way of obtaining this information is through an interactive dialogue in which the user fills out the "form" step by step, by responding to questions about the problem he is specifying. One benefit of this approach over an actual form is apparent in situations where the consequence of indicating one choice implies that another section must be completed. No manual method can guarantee the user will complete the form correctly, but in an interactive environment, all relevant sections can automatically be presented as a dialogue sequence. Conversely, many columns, or even entire forms, may be unnecessary
Disadvantages

There have traditionally been several disadvantages to the dialogue approach to offset the advantages discussed above. Often, interactive dialogue mode has been considered simply as a crutch for beginners, to be discarded as soon as they have learned the series of steps through which the dialogue is taking them.

What are some of the problems that make people unwilling to tolerate dialogue once they have used the same sequence a few times? Some contributing factors are:

- **Type of Device**—Most dialogue up until now has been conducted on typewriter devices, in particular the Teletype Model 33. These devices are slow and noisy. In addition, they are linear devices and not well adapted to two-dimensional presentation.
- **Speed of Display**—This is related to the type of device; but even if a buffered CRT such as a Model 100 Uniscope® terminal is used for dialogue, it will be limited by the throughput of the telecommunication line.
- **Response Time**—Most present-day dialogue driven programs reside on large, centralized systems which time-share the dialogue user with many other processes. Usually, each step in the dialogue incurs communications, processor and operating system overhead plus disc access time, all of which can amount to a noticeable delay between steps of the dialogue which may be irritating to an experienced user.
- **Dialogue Design**—Many dialogues appear to be poorly designed for their intended audience. It is common to encounter stylistic extremes ranging from verbose instructions intended to be helpful but ending up making the user feel the dialogue designer assumes he is an idiot, or alternately of cryptic messages and symbols that reinforce the user's feelings of confusion and mystery when using the computer. Other faults are a poor ordering of questions, designed more for the benefit of the machine than the user, and poor "sign posting," so that the user can easily lose his way in a series of questions, or suddenly find unexpected things happening.
- **Cost**—Typically, dialogue driven processes have required large systems, and have consumed large amounts of memory and other system resources. Telecommunication lines, particularly with high throughput, are another heavy expense. If one takes a narrow view and simply counts tangible costs, dialogue driven processes are not an "efficient" way of utilizing resources. Lastly, the development cost of software for on-line applications has been considerably greater than the cost of developing equivalent applications running in a non-interactive environment.

**DIALOGUE APPROACH**

Our approach to finding solutions to the ease of use requirement is essentially dialogue based. Many of the problems discussed above are tending to be diminished by current technological trends. For example, CRT terminals are no longer considered exotic, high cost items; similarly, communications technology is tending towards higher capacity at less cost. The possibilities offered by intelligent terminals suggest a new distribution of functions, outside of the host system. In such an environment it is clear that at least a part of the work associated with managing dialogue could be performed by the terminal itself.

While technological advances are tending to eliminate most of the disadvantages associated with the mechanics of computer dialogues cited above, we still see one serious problem remaining—that is the design and implementation of effective dialogues. Our approach has been to separate the dialogue activity from the system that is to be dialogue driven, by creating some specialized software: a Dialogue Specification Language, which encodes a dialogue from a high-level description, and the Dialogue Processor, which is a run-time interpreter, managing the interactive process of presenting the steps of the dialogue to the user and capturing his responses.

In a typical present-day situation, the dialogue is inextricably embedded in the application or system program. The form of the dialogue is entrusted to the software designer, who is frequently not an expert in human communications. Ideally, we believe, dialogues should be written by skilled communicators, such as technical writers, educators, industrial psychologists or specialists in the subject matter of the dialogue. Dialogue design should be made more responsive to human needs; the format, presentation and choice of words should reflect the user's familiarity with the interactive process. The terminology used should vary to suit different user communities, for example, people whose native tongue is not English.

The above requirements suggest that, for a given application, many dialogues could exist simultaneously, each turned to a different user need. The Dialogue Specification Language and Dialogue Processor were designed to meet these goals.

**DIALOGUE PROCESSOR**

The Dialogue Processor is a program that supervises an interactive process in which a succession of messages is displayed on a CRT screen. For each message the user's response is obtained, analyzed and used to control succeeding interactions. The Dialogue Processor is a general purpose program; it will operate with any dialogue that has been suitably encoded. In addition to the display output described above, the Dialogue Processor produces two
other outputs: firstly an audit file from which a listing of the dialogue, expressed in natural language style, can be extracted for the user, and secondly a formatted output file that embodies the essential information of the user's responses.

The Dialogue Processor is essentially one component of a two-part system, the other component consisting of the process that is to be dialogue driven. Figure 1 is a block diagram of the Dialogue Processor; it depicts the output extracted from the user's responses as a string of parameters that might drive one of many different processes, such as an application customizer, operating system or file management system.

The questionnaire of Figure 1 is expressed in Dialogue Specification Language and encoded by the DSL compiler. It is worth noting that in general the writer of the dialogue will not be the same person as the interactive user, the former will need some specialized skills, while the latter is assumed to be non-technical.

Choice trees

Consider a dialogue as represented by a tree, with nodes corresponding to points at which the user has a choice and branches corresponding to messages and user input. The whole tree corresponds to the dialogue structure, and paths through the tree are permissible sequences of user responses. The dialogue can be conceptualized as having a syntax which can be represented with the following metalinguistic symbols:

{   } [   ]   \ldots

The following example illustrates the steps by which we can go from a syntactic structure to a dialogue. Figure 2 shows the syntax for the EXTENT statement in the JCL for Sperry Univac's OS/3 Operating System. Using the format and the accompanying rules as set out in the Reference Manual, the syntax can be expressed rigorously in the format shown in Figure 3. Figure 4 shows the substitution of messages for mnemonics and codes. If we were to make a dialogue in which each group of parameters in turn were to be offered to the user, we could represent the syntax internally as the tree shown in Figure 5. Here the nodes exercise the following control over the branches which emanate from them:

K = Sequential—The Dialogue Processor follows the branches one by one in sequence (the user does not have control).

A = Mandatory Exclusive—The user must choose one and only one branch. The Dialogue Processor follows the path indicated.

B = Optional Exclusive—The user may choose one branch, which the Dialogue Processor will follow; alternately, he is permitted to signify a null choice—"none of the above."

We emphasize it is the Dialogue Processor that enforces the choice rules at each node, making it impossible for the user to violate them. Thus, any path the user is able to trace through the tree results in a syntactically correct EXTENT statement.

As we will see later, the Dialogue Specification Language has features enabling us to write dialogues embodying the syntactic structures illustrated above, directly.
Functions of the dialogue processor

The Dialogue Processor is responsible for the presentation and sequencing of dialogue elements. Dialogue elements consist of specific messages to the user (concerned with the subject of the dialogue); standard messages to the user (concerned with control, prompting, error notification); requests for the user to specify choice (menu selection) and request for the user to enter data. For each step in the dialogue, the Dialogue Processor provides the appropriate mechanism for the user's response. Thus, the user may be confronted with a list of elements from which to choose; accompanying the list will be instructions on how to signify his choice and what rules govern this particular choice. Choices may be mandatory or optional, mutually exclusive, or inclusive. In each case the Dialogue Processor automatically enforces the choice rule. Each interaction is terminated by the user signifying choice, acknowledgment, or by entering data. Following verification that the user's response was valid, the Dialogue Processor proceeds to the next interaction called for by the dialogue structure. Deciding where to go next is a Dialogue Processor function which will, in many cases, be dependent on the previous response. As the user follows a path through the dialogue, he may discover he has taken a "wrong turn" or may wish to review and change some earlier response; a feature of the Dialogue Processor is the ability to back up or assist the user to recover from an unintended response.

DIALOGUE SPECIFICATION LANGUAGE

DSL is a high level language especially designed for the creation of dialogues which will run under the control of the Dialogue Processor. It has facilities for specifying the dialogue structure, messages to be displayed, input to be entered by the user and the content, format, and mapping rules for both the formatted output file and the natural language report output. DSL source programs are compiled into a mixture of tables and interpretive code—the output of the DSL compiler is termed the encoded dialogue. The Dialogue Processor is the run-time interpreter for encoded dialogues.

A reference manual for DSL has been documented elsewhere. In this paper we will only attempt to highlight some of the unusual features of the language.

Trees

DSL has the facility to describe trees which are used to represent dialogue structures. The language has a tree declaration statement, the syntax of which is shown in Figure 6. The DSL tree is essentially a specialized program structure consisting of an optional subroutine called the trunk, a control mechanism called the node type, and a collection of subroutines called the branches. DSL possesses a unique command

PRESENT tree

which, when executed, starts an interactive process in which the Dialogue Processor displays messages and elicits responses to enable it to traverse the tree under guidance from the user.

Associated with each branch are branch actions. These actions may display messages, request input or perform output and other processing functions. At the time the branch is initially displayed, all the branch actions are performed on a temporary basis; only after the branch has been selected are the actions made permanent. A branch may terminate with another tree, thereby providing a nesting structure.

Control of execution of the branch actions is governed by the node type of the tree. For an exclusive node, all the branches are executed temporarily, then the one selected by the user has its actions made permanent. An inclusive
Figure 5—Tree structure for EXTENT statement dialogue
node has a similar effect, except that the branches that remain unchosen after the first interaction are again presented to the user. In the case of a sequential node, only one branch at a time is executed. The purpose of a sequential node is to present branches in order. Parallel nodes are used for data entry sequences; neither parallel nor sequential nodes require a choice to be exercised by the user. The reject node type is the inverse of an inclusive choice—in this case it is those branches the user does not indicate that are executed. If a node type is qualified as optional, the user can indicate that he does not want to select any branch.

Branches and trees can be either written in line, or declared and named. If a tree is named, multiple references to it are possible, enabling a dialogue structure that is a network to be easily represented in DSL.

The trunk subroutine is executed once only, on presentation of the tree; its intended purpose is to display a heading for the user.

The optional MASKED BY and SELECT clauses will be explained following the discussion of DSL masks, below.

**Data items, arrays and masks**

DSL has facilities for declaring data items and arrays. Associated with every array is a mask with a bit corresponding to each element of the array. The effect of the mask is to only allow the DSL program to see those elements of the array for which the corresponding mask bit is set to 1.

If an array is referenced in a branch subroutine, it will have the effect of replicating the branch once for each element of the array; thus the array reference in the \( n \)th instance of a replicated branch is implicitly a reference to the \( n \)th element of the array. The above mechanism can be used to display all the elements of an array to the user as the branches of a tree, and, through the use of the select facility that exists in DSL, the array's mask bits can be set to mask off those elements that correspond to the branches not chosen.

Masks can also be used to condition the branches of a tree, this is the function of the MASKED BY clause. If this clause is used, the \( n \)th branch of a tree will only be presented to the user if the \( n \)th bit of the mask is set to 1.

**Block structure**

DSL uses a block structure to control looping, initialization and the scope of variables. The syntax for block declarations is shown in Figure 7. Looping may be controlled by a count, or continue while or until a certain condition is true.

Arrays can also be used to control looping. Using the DO FOR EACH declaration, a program block
controlled by an array will iterate through each element of the array for which the corresponding mask bit is set to 1. Any reference to the controlling array within the block is actually a reference to the current element of the array. This mechanism provides an implicit subscript feature in DSL.

The case statement has its conventional meaning, but the CONTROLLED BY mechanism provides a powerful extension of case—the effect of this construction is that for each 1 bit set in the mask of the controlling array, the corresponding general statement is executed.

The block mechanism is also used to control initialization. Variables are automatically initialized on entering, or re-entering the block in which they are declared. The scope of a variable is the block in which it is declared and all nested blocks.

Other features of DSL

There are powerful formatting facilities in DSL which can control the way data appear in the various outputs from the Dialogue Processor. Facilities include decimal placement and/or suppression, left or right justification, zero fill and centering.

DSL is also a general programming language with arithmetic, logical expression and string manipulation capabilities. The language has a simple macro facility.

The DSL programmer does not have to write explicit routines for handling user choices—the display of standard control and error messages, the positioning of headings and the tabulation of choices are all done automatically; similarly, the programmer does not have to concern himself with writing backtracking or error recovery procedures, as these are built in to the Dialogue Processor.

OUTPUTS OF THE DIALOGUE PROCESSOR

As the user progresses through a dialogue, he is building two distinct outputs. (We are not including, in this discussion, output messages to the CRT.)

Target file

As previously observed, the Dialogue Processor is one component of a two-part process; it is essentially the front-end to some dialogue driven process. The target file consists of exactly the input required by the complementary process. The Dialogue Processor is responsible for formatting the input correctly as well as ensuring a syntactically correct stream of data to the dialogue driven process.

The input requirements of many proposed target processes are machine-oriented rather than designed with good human factors (for instance, the cryptic codes and rigorous punctuation rules of JCL). One important function of the Dialogue Processor is to insulate the user from such problems, and to do so, it possesses a powerful mapping capability.

Figure 8 shows the syntax of a dialogue that is part of the information needed to specify a payroll. As the user responds to each element of this dialogue, he is specifying a rule that pertains to his payroll program; in this particular example, the rules for overtime pay. The use to which this information will be put is to furnish parameters to a customizer which will adapt a generalized payroll package to the user’s particular methods. The input requirements of the customizer are, in fact, a collection of short procedural statements in a fixed assembly language-like format which are interpreted at run time. Figure 9 shows the paths selected during two passes through the tree, and beneath each statement is the code needed by the customizer to cause the processing to be carried out in the way described. The Dialogue Processor is able to map a human-oriented statement of requirements to the machine-oriented version required by the target process.

Report document

The other output created by the Dialogue Processor is an audit file from which a user oriented report document is extracted. Essentially the report summarizes, for the user’s benefit, the dialogue that was transacted. Figure 10 is an
SCREEN 100, OVERTIME PAY CALCULATION FOR employee-type EMPLOYEES

NO OVERTIME PAID TO THIS EMPLOYEE TYPE

OVERTIME IS PAID USING THE FOLLOWING RULES:

ONLY AFTER min-hrs REGULAR HOURS WORKED,

MAXIMUM NUMBER OF OVERTIME HOURS PERMITTED = max-hrs,

NO MINIMUM NUMBER OF REGULAR HOURS,

UNLIMITED OVERTIME HOURS PERMITTED.

FIXED FOR ALL OVERTIME HOURS @ fixed-rate × (HOURLY RATE)

RATE IS VARIABLE: FOR FIRST hrs-1 HOURS @ rate-1 × (HOURLY RATE),

FOR NEXT hrs-2 HOURS @ rate-2 × (HOURLY RATE),

FOR NEXT hrs-3 HOURS @ rate-3 × (HOURLY RATE),

FOR REMAINING HOURS @ rate-4 × (HOURLY RATE)

Figure 8—Syntax for overtime rules in payroll dialogue

example of part of the report produced in specifying the payroll that was discussed in the previous example. The text of this report can be identical with the messages associated with each path selected during the interactive process (as is the case with Figure 10), or it can be separately specified to provide any suitable documentation. It should be noted that the style of the report can be close to a natural English description of the problem and thus be readily understandable to a non-technical person.

DIALOGUE EDITOR

The purpose of the Dialogue Editor is to enable the user to change his responses to a dialogue that has already taken place. The user may wish to change the data that he previously submitted in response to a data entry request, or he may wish to change a previously made choice selection.

The editor is intelligent in the following sense: Certain changes that the user can make will affect the subsequent dialogue, e.g., the deletion of a branch should result in all nested branches also being deleted. The editor will attempt to effect all consequent changes automatically, however, where the result of an earlier change lead to a situation where the user must provide more information, the editor solicits this via the corresponding dialogue.

Audit file

The editing mechanism is based on an audit file which is produced during a dialogue session and which contains essentially two kinds of information: firstly, the data that goes into the Report Document, which is user visible and is the means through which the user signifies any changes, and secondly, control information which is not user visible, but which is a trail of everything the user did during the session. The Dialogue Professor always operates with four files, Figure 11. The user indicates which paragraphs he wishes to change by writing the paragraph numbers to the change file. During an editing session, the Dialogue Processor reads unchanged paragraphs from the old audit file, produced at the previous session, interpreting the control information in the same way as if it had come from the keyboard. When it reaches a change paragraph, it displays the old version on the CRT screen, and waits for the user to...
SCREEN 100, OVERTIME PAY CALCULATION FOR TYPE 2 WEEKLY EMPLOYEES:
OVERTIME IS PAID USING THE FOLLOWING RULES: ONLY AFTER 40.0 REGULAR HOURS WORKED, MAXIMUM NUMBER OF OVERTIME HOURS = 40.0, RATE IS VARYING:
FOR FIRST 20.0 HOURS @ 1.5X HOURLY RATE, NEXT 10.0 HOURS @ 2.0X HOURLY RATE, REMAINING HOURS @ 2.5X HOURLY RATE

---

WW.X2 REGHRS=40.0 #WWXXU2#RETURN#WWXXU2
WW.X2TREGHRS+OTHRS =SSAVEA140.0 #WWXXS#WWXXR#WWXXR
WW.X2S$SAVEA=0.0 #OTHRS =140.0 #REGHRS#WWXXU2
WW.X2XREGHRS#REGHRS#WWXXR
WW.X2U2OTHRS =140.0 #WWXXJ2#WWXXT#WWXXR
WW.X2J240.0 #OTHRS #WWXXT2
WW.XT2OTHRS -20.0 =SSAVEA10. #WWXR2#WWXXUP#WWXR2
WW.XR20. =SSAVEA#WWXR2
WW.XQ2OTHRS =SSAVEA=OTHRS #RATE =1.5 #OTPAY #WWX12
WW.X125SAVEA=10.0 =SSAVEB10. #WWXRN2#WWXYM#WWXR2
WW.XRN2. =SSAVEA#WWXR2
WW.XT25SAVEA=SSAVE=SSAVEA#RATE =2.0 #OTPAY #OTPAY #WWXYG2
WW.XQ25SAVEA=SSAVEA#RATE =2.5 #OTPAY #OTPAY #WW.R2

**********

SCREEN 100, OVERTIME PAY CALCULATION FOR TYPE 3 HOURLY EMPLOYEES:
OVERTIME IS PAID USING THE FOLLOWING RULES: NO MINIMUM NUMBER OF REGULAR HOURS, MAXIMUM NUMBER OF OVERTIME HOURS = 20.0, RATE IS FIXED FOR ALL OVERTIME HOURS @ 1.5X HOURLY RATE

---

WW.X3 OTHRS =120.0 #WWXXJ3#WWXXT3#WWXXT
WW.X3J3=OTHRS #WWXXT3
WW.XT3RATE =OTHRS #1.5 #OTPAY #WW.R3

Figure 9—Customizer code for two sets of overtime rules

indicate the desired change. If a choice change is requested, the user is shown the original menu at that point, and the Dialogue Processor takes further input from the keyboard until the user terminates that particular request. Following a change, the information on the old audit file may or may not be valid. Whenever a conflict is detected, the Dialogue Processor returns to the user for responses that will resolve the inconsistency.

APPLICATIONS FOR THE DIALOGUE PROCESSOR

The initial application for the Dialogue Processor is in conjunction with a customizer that has been developed for use with packages for the small business machine market. The Dialogue Processor will replace the questionnaire forms that have been used with previous customizers.

Both the Dialogue Processor and customizer can run in the user's own machine; thus, it will be possible for the users to create their own applications without knowledge of a computer language. The automatic documentation feature, in conjunction with the Dialogue Editor, will also enable users to modify and maintain their own applications.

We are currently investigating other areas where the Dialogue Processor should prove effective. Some of these applications are:

- Formulation of commands by users in the traditionally difficult command language areas such as Job Control, System Generation, Network Specification and Control, and other instances where the user is required to submit complex specifications to a process.
- Creation of on-line applications (such as financial, medical, etc.) where the dialogue for receiving user requests and transactions has traditionally been an integral part of the application. We will attempt to demonstrate the advantages of flexibility and lower development costs the Dialogue Processor will provide.
SCREEN 10A, OVERTIME PAY CALCULATION FOR TYPE 1 SALARIED EMPLOYEES:
NO OVERTIME PAID TO THIS EMPLOYEE TYPE

SCREEN 10B, OVERTIME PAY CALCULATION FOR TYPE 2 WEEKLY EMPLOYEES:
OVERTIME IS PAID USING THE FOLLOWING RULES: ONLY AFTER 40.0 REGULAR
HOURS WORKED, MAXIMUM NUMBER OF OVERTIME HOURS = 40.0, RATE IS VARIABLE:
FOR FIRST 20.0 HOURS $1.5 X HOUlRY RATE, NEXT 10.0 HOURS $2.0 X
HOURLY RATE, REMAINING HOURS $2.5 X HOURLY RATE

SCREEN 10C, OVERTIME PAY CALCULATION FOR TYPE 3 HOURLY EMPLOYEES:
OVERTIME IS PAID USING THE FOLLOWING RULES: NO MINIMUM NUMBER OF REGULAR
HOURS, MAXIMUM NUMBER OF OVERTIME HOURS = 20.0, RATE IS FIXED FOR ALL
OVERTIME HOURS $1.5 X HOURLY RATE

Figure 10—Part of the payroll report output

- Creation of end user facilities for interfacing with data base management systems; in particular, using dialogues to enable non-technical users to generate valid queries or transactions against data bases having complex structures and interrelationships. We plan to experiment with a variety of styles of interaction appropriate to different users, which the Dialogue Processor will map to a standard interface with the DBMS.
- Creation of dialogues for engineering diagnostics, which would enable the Dialogue Processor to be used by the customer to step through a series of tests and produce a detailed report of the fault, prior to calling the customer engineer.

We have already started to investigate some of the above areas. It is obvious there are many other applications for the Dialogue Processor.

CONCLUSION

In the first part of this paper we have identified some of the issues that affect ease of use. Particularly in the area of
business applications, we can see the emergence of certain trends: Application generators, problem oriented languages, interactive systems and some serious effort to develop systems that employ natural language.

To aid in the development of easy-to-use systems we have created the Dialogue Processor and Dialogue Specification Language. With these tools we believe the utility of interactive dialogues can be greatly extended.

The Dialogue Processor is seen as one component of a two-part system. Since many different processes can be dialogue driven, the Dialogue Processor is truly a general purpose facility. Initially, the Dialogue Processor will interface with existing systems; however, once proven, we can expect the development of new systems especially designed to operate with a dialogue front-end.

With the Dialogue Processor, a user can express complex commands and specifications without being burdened with the effort of learning a programming language. All communications can be conducted in natural language; and furthermore, syntactic considerations are handled automatically.

DSL will encourage the development of effective dialogues tuned to users' needs. Dialogues can be developed as a separate activity from the processes they drive.

The Dialogue Processor has been designed with very modest processor and memory requirements; it is, therefore, feasible for it to reside in a programmable terminal. With this arrangement, the user will obtain very quick response between successive steps in the dialogue so it may be acceptable to use dialogue as the regular mode of expressing commands, and not simply to regard it as a training aid.

We expect the Dialogue Processor not only to make computers more accessible to non-programmers but also to help computer professionals become familiar with new facilities more quickly and use them more effectively.

ACKNOWLEDGMENTS

The author is greatly indebted to Mr. Jerome Holland for many discussions and insights, without which the work described in this paper would never have become a reality. Acknowledgment is also due for the efforts of Messrs. David Monk and William Overholt of Sperry Univac, who first saw the relevance of the Dialogue Processor to customizing small business applications, and who were responsible for supporting the research effort and its implementation in the Sperry Univac BC/7. Lastly, a large debt of thanks is due to Mr. Theodore Bonn for his encouragement and support and to numerous colleagues at the Sperry Research Center for their invaluable technical assistance.

REFERENCES