Exhibit 1016 – Part 2

reporting errors immediately may be distracting but immediate confirmation of a coded value as a result of a validation may also be desirable. As a rule of thumb, in cases where the input comes from a source document, validation is deferred to the end of the form to avoid disrupting the keying process; where there is no source document, validation is immediate.

If any errors are encountered, the dialogue does not redisplay a new blank form; the form with the previous answers is shown with any erroneous items indicated. In the human–human analogy, you only get a new form if you have made such a complete mess of the previous one that it is simpler to start again from scratch.

As with the Q&A and menu structures, an individual answer may represent either selection from a list of possible replies or an arbitrary data value. It is traditionally entered as a text string but, as we saw in Chapter 4, a selection input may be made by scrolling round a supressed menu; this is analogous to 'delete as appropriate' on clerical forms.

5.2.2 Design Criteria

Forms are a natural mechanism for the entry of transaction data since a transaction, by definition, comprises a relatively standard set of data values. Thus, the structure is particularly appropriate where the source of the data is an existing clerical form. In the example of Fig. 5.1, the dialogue knows in advance precisely what data items are required since a field is defined in terms of content, slot and attributes.

The criteria discussed in Section 4.3.2 for Q&A apply to the phrasing and display of the individual questions on the form. Answer fields should be clearly differentiated from the rest of the form by boxing with a particular attribute or by delimiting them with 'decorators' such as the brackets in the example of Fig. 5.1; the remainder of the screen should be *protected* so that input echo is restricted to these areas. The fact that the screen contains a number of questions imposes additional requirements which complicate the design of the screen layout; layout considerations are discussed in Chapter 7.

It is essential that the form displayed on the screen should match any clerical form from which information is to be taken. It need not have exactly the same physical appearance, indeed this can result in a very cluttered screen, but all input items must appear in the *same relative order* and have the *same format* as the source data. The reasons for this are obvious if you consider the input process.

Often, all the items required cannot be displayed on a single screen but must be broken into sections which are displayed on successive screens. It is essential that this breakdown preserves a *logical grouping* so that related items are not split over different screens. An existing clerical form will provide strong clues as to where such breaks should occur; Chapter 7 provides an example of this process.

The forms structure is highly supportive. Where a physical form provides the source of the input, little additional support is required from the dialogue. Since the user is required merely to transcribe the information, the screen can be relatively full, with the questions merely terse captions. The minimum requirement is that the user can read his input to check it.

Where there is no corresponding physical form, a user will read the screen form to determine the input required. The screen must therefore be less full and the questions more explanatory. Help and error messages should be specific to a particular question and the user should be able to request help in response to any question on the form. Support can also be provided by including format details in the questions or answer fields, e.g.

Invoice Date (dd/mm/yy)[ ] or Invoice Date [/ /]

It is important not to over-format the input in such cases; the address format

No & Road [ ]
District [ ]
Town [ ]
County [ ]

admits only of a typical suburban address whereas address styles vary widely depending on location. With one particular public utility, it is almost impossible to book a service if you live on a farm — their computerised booking system insists that every house has a number and will not proceed further until the number is input!

A potential problem occurs where a clerical form contains additional fields which are not required to be input. Such areas on the clerical document should be clearly distinguished from those containing data to be input. A clerical form may also contain optional items, or items which are completed only if a given value is entered in an earlier field; ideally, these should occur at the bottom of the form. If this is not possible, the system should automatically skip over any question, corresponding to these optional items, which does not require an answer and should provide confirmation that it has done so.

SKYHAWKE Ex. 1016, page 44

5.2.3 Implementation

Like the menu structure, the form structure has two main stages: a single display of the form followed by repeated requests for input until the form is complete. The form may be complete either when the user enters a specific form terminator or, if no such terminator is defined, when all the questions have been answered.

A form can be defined as a set of output fields. Unlike a menu, however, there is no single way of subdividing this set into subsidiary structures which will define any form. There may be fields which, in some forms, perform the role of a 'form header' or 'form trailer'; there are fields in Fig. 5.2 which obviously fulfil this role. There are other output fields which specify captions; although these represent questions to the user, they have no significance to the dialogue process. There are also various other outputs such as subheadings and 'decorators' which define the form's outlines. We might consider all these output fields to be 'background' items which define the shape of the form.

Thus a form can be described by a data structure of the type described in Fig. 5.3. Note that a simple boxing of the form could be accomplished merely by adding an attribute, *boxed*, to the FormDIB definition; see Question P1 of Chapter 4. Our FormFieldListType definition allows a more flexible use of decorators.

Field Definition

Content: [ ]

Slot:

row [ ] column [ ] width [ ]

Attributes:

Foreground [ ] Background [ ]

Bold (y/n) [ ] Blink(y/n) [ ]

F1 = Accept Form F2 =help ESC = abort

Fig. 5.2. An input form.

```
FormFieldListType = array[1..MaxFormFields] of FieldType;
FormDIBtype = record
              NumberOfFormFields  : byte;
              FormFieldList       : FormFieldListType;
              end;
```

Fig. 5.3. A form Dialogue Information Block (FormDIB).

SKYHAWKE Ex. 1016, page 45

The form in Fig. 5.2 can then be created by the program fragment:

```
with FormDIB do
   begin
   FormOutline:='back=blue,fore=white';
   TLCorner:=chr(201);    TRCorner:=chr(187);
   BLCorner:=chr(200);    BRCorner:=chr(188);
   horizontal:=chr(205); vertical:=chr(186);
   bar:=''; spaces:='';
   for k:=1 to 47 do
      begin
      bar:=concat(bar,horizontal);
      spaces:=concat(spaces,' ');
      end;
   NumberOfFormFields:=15;
   CreateField(FormFieldList[1],concat(TLCorner,bar,TRCorner),
               2,10,49,FormOutline);
   CreateField(FormFieldList[2],
               concat(vertical,
                      '              Field Definition               '
                      ,vertical),
               3,10,49,FormOutline);
   CreateField(FormFieldList[3],concat(vertical,spaces,vertical),
               4,10,49,FormOutline);
   CreateField(FormFieldList[4],
               concat(vertical,
                      '   Content: [                              ] '
                      ,vertical),
               5,10,49,FormOutline);
   ..............................................................
   CreateField(FormFieldList[15],concat(BLCorner,bar,BRCorner),
               16,10,49,FormOutline);
   end;
```

The form can be displayed on the screen by the procedure DisplayForm illustrated in Fig. 5.4.

```
procedure DisplayForm(FormDIB:FormDIBtype);
var k : byte;
begin
ClearScreen;
with FormDIB do
   begin
   for k:= 1 to NumberOfFormFields do
      DisplayField(FormFieldList[k]);
   end;
end; {DisplayForm}
```

Fig. 5.4. Defining and displaying a form.

What parameters are required to define the 'foreground' of the form? The foreground represents the answers which the user supplies to the questions on the form. The display of the questions themselves has been taken care of implicitly in the form background; as with the menu structure, a question will not normally be redisplayed when a particular answer is requested. There is no possibility of implementing a form in 'teletype' mode. However, it is conceivable that some indication, other than the input echo, of the current question may be displayed. For example, the question or a subheading might be highlighted. We will allow the possibility of an explicit question when each answer is requested.

Thus, to process the user's answer to the question requires the same parameters as in the Q&A or the menu structures, i.e. those parameters which were specified in the QandA_DIB data structure. Since the values of these parameters are likely to be different for different questions, there will be a separate QandA_DIB for each question within the form. The foreground of the form is represented by a set of QandA_DIBs as in Fig. 5.5.

```
QandAListType    = array[1..MaxQuestions] of QandA_DIBtype;
QandASet_DIBtype = record
                   NumberOfQuestions : byte;
                   QandAList         : QandAListType;
                   CurrentQuestion   : byte;
                   end;
```

Fig. 5.5. Dialogue Information Block for form input.

In practice, there are more efficient storage mechanisms than the structures comprising arrays of arrays of fields which we have used, e.g. arrays of pointers or linked lists. We will continue to use array structures for list structures in the descriptions because of their conceptual simplicity.

A ControlDIB must be initialised with values which control how the input will be handled by the input/output processes for each Q&A step; the need for consistency dictates that these values will be the same for all items on the form. Additional control characters are needed to allow the user to skip over a question, to go back to a previous question and to hand the completed form back to the dialogue process. These values occupy three of the previously undefined elements of the ControlDIBtype data structure as illustrated in Fig. 5.6.

PointNextField and *PointPriorField* allow the user to scroll around the questions on the form; *AcceptForm* provides a facility for the user to confirm completion of the form by pressing a specific key. If the ReadField process receives any of these characters, it terminates immediately and returns the

SKYHAWKE Ex. 1016, page 47

```
ControlDIBtype = record
                 ControlBuffer     : byte;
                 rubout            : byte;
                 EchoSwitch        : OffOn;
                 AcceptField       : byte;
                 RequestAbort      : byte;
                 RequestHelp       : byte;
                 PointNextTarget   : byte;
                 PointPriorTarget  : byte;
                 AcceptTarget      : byte;
                 PointNextField    : byte;
                 PointPriorField   : byte;
                 AcceptForm        : byte;
                 reserved4         : SetOfByte;
                 end;
```

Fig. 5.6. The ControlDIB with forms controls.

control character in ControlBuffer so that the dialogue process can take any necessary action. If both PointNextField and PointPriorField are null, the user is constrained to answer the questions in order; if AcceptForm is null, the dialogue will assume that the form is complete when the last question in the set has been answered.

The user also proceeds to the next question when he completes the current one. A question can be completed in two ways: *manual skip* requires the user to enter an explicit terminator character for each answer field; *auto skip* proceeds to the next question as soon as the answer field slot is filled. The ReadField process supports both these mechanisms. If ControlDIB contains a non-null value for AcceptField, this value must be entered to terminate input — a manual skip mechanism. If AcceptField is null, the input will terminate when answer.slot.width characters have been entered — an autoskip mechanism.

Manual skip is to be preferred unless the vast majority of the input is of a fixed length, in which case the need to type fill characters is minimal. What is to be avoided at all costs is a combination of the two, i.e. where a skip to the next field occurs either when the last input position is filled or when a skip character is input. Users will get into the habit of typing a skip character at the end of each input and will do so even where it is of maximum length, causing a double skip. This inconsistency is precluded with our single ControlDIB which controls all input throughout the form. There is another problem with auto skip which concerns the editing of input: if the user is going to make a typing mistake, he had better do it before the last character!

A procedure, FormInputIV, which performs immediate validation of any input which represents selection from a targetlist is illustrated in Appendix I.

```
procedure FormInputDV(var QandASet_DIB:QandASet_DIBtype;
                      var ControlDIB:ControlDIBType);
var complete : boolean;
begin
with ControlDIB,QandASet_DIB do
   begin
   complete:=false;
   repeat
      ArbitraryData(QandAList[CurrentQuestion],ControlDIB);
      QandAList[CurrentQuestion].ErrorFlag:=off;
      if equal(ControlBuffer,RequestAbort)
      or equal(ControlBuffer,AcceptForm)
      or ((CurrentQuestion=NumberOfQuestions) and (AcceptForm=0)) then
         complete:=true
      else  {test for scroll back}
      if equal(ControlBuffer,PointPriorField) then
         if CurrentQuestion=1 then CurrentQuestion:=NumberOfQuestions
                              else CurrentQuestion:=CurrentQuestion-1
      else  {test for scroll forward}
      if equal(ControlBuffer,PointNextField) then
         if CurrentQuestion=NumberOfQuestions then CurrentQuestion:=1
                              else CurrentQuestion:=CurrentQuestion+1
      else {proceed to next outstanding question}
         begin
         while (CurrentQuestion<=NumberOfQuestions)
         and   (QandAList[CurrentQuestion].ErrorFlag=off) do
            CurrentQuestion:=CurrentQuestion+1;
         if CurrentQuestion>NumberOfQuestions then
            if AcceptForm=0 then complete:=true
                            else CurrentQuestion:=1;
         end;
      if not equal(ControlBuffer,RequestAbort) then
         ControlBuffer:=0; {because it has been actioned}
   until complete;
   end;
end; {FormInputDV}
```

Fig. 5.7. Input to a form with deferred validation.

We will consider here a procedure, FormInputDV illustrated in Fig. 5.7, which performs deferred validation. If validation is to be deferred until the form is completed, each input must be treated as an arbitrary data value, i.e. each input field is accepted via procedure ArbitraryData (Fig. 4.5).

When the FormInput procedure returns, the user's response to question k will be contained in QandAlist[k].answer.content. The dialogue must then invoke validation of each of these answers and, if an error is encountered, set the ErrorFlag and ErrorMessage in the QandA_DIB for QandAList[k].

This procedure is called repeatedly until all errors are eliminated as illustrated by the following program fragment:

```
for k:=1 to NumberOfQuestions do
   begin
   {ensure each question is asked the first time through}
   {it's an error initially since he hasn't entered anything}
   QandAList[k].ErrorFlag:=on;
   {but there is no error message to display}
   QandAList[k].ErrorMessage.content:=' ';
   end;
```

SKYHAWKE Ex. 1016, page 49

```
CurrentQuestion:=1;
complete:=false;
repeat
   FormInputDV(QandASet_DIB,ControlDIB);
   if equal(ControlBuffer,RequestAbort) then
      complete:=true
   else
      begin
      NoErrors:=true;
      for k:=1 to NumberOfQuestions do
         begin
         {validate answer k
          if invalid then
             set ErrorFlag:=on for QandAList[k]
             set ErrorMessage content for QandAList[k]
             if NoErrors then
                NoErrors:=false
                CurrentQuestion:=k}
         end;
      if NoErrors then  complete:=true;
      end;
until complete;
```

Note that the process illustrated in Fig. 5.7 allows the user to change any of the answers, not just those that were flagged as invalid. This reflects clerical form filling; if you are handed back a form to correct, you can change anything on it. By default, the dialogue will request the next outstanding answer — the next one with ErrorFlag set on. To ensure that all answers will be requested the first time through, the dialogue initially sets the ErrorFlag to 'on' in each question.

5.2.4 Summary

Many interactive systems have a requirement for data entry via a standard sequence of questions. The form structure suits such usage, which is common in accounting and order processing applications. It is quicker than Question and Answer, it can handle a wider range of inputs than menus, and it can be used by any level of user. Most people are familiar with the concept of filling in forms even if they claim to abhor doing it! Because it has a sequential rather than a tree structure it is less appropriate for option selection.

Another area where form filling has been used is to specify the parameters for querying databases. The mechanism is called *Query by Example*. The fields on which the database can be searched are displayed as column headings on a form. The user enters in each column the values on which he wishes to search. The values input in a given column are 'ORed' and the columns are 'ANDed', so that an input of:

Grade	Location	Skill
SEC	LON BIR MAN	A C

would select all SECretaries in LONdon, BIRmingham or MANchester and who are classed as having skills A or C. It eliminates the problems caused when parentheses must be used to cater for different precedence in the logical operators, e.g.

(grade=SEC) and ((loc=LON) or (loc=BIR) or (loc=MAN))
and ((skill=A) or (skill=C))

Multiple choice menus are also a type of form filling. With such a menu, the user is presented with a list of options but is not restricted to a single selection; he may make zero or more selections from it. Figure 5.8 illustrates such a menu which is presented to the user of a Terminal Emulator package, a program which enables a microcomputer to be used as a dumb terminal. Each of the options on the list represents a terminal characteristic which may be set or unset by the user. To change a characteristic, the user scrolls to the relevant option and types 'y' to set it, or 'n' to unset it. Thus the menu illustrated in Fig. 5.8 is effectively a form with five questions, each of which expects a yes/no answer; the questions have a default answer and so he need not answer each explicitly. This structure is typical of multiple choice menus. Note that, as with most forms, the user can continue to scroll around the options until he presses the AcceptForm key, function key 1 in this case.

Terminal Configuration	
Local Echo	[Y]
Local Print	[N]
Auto Linefeed	[Y]
Auto LineWrap	[N]
VT100 Emulation	[Y]
press F1 to exit	

Fig. 5.8. A multichoice menu.

5.3 The Command Language Structure

5.3.1 Features

The *command language* structure is almost as commonplace as the menu structure, primarily because it has been a very common style for computer operating systems. It is at the opposite end of the spectrum from the menu structure. The use of the term 'command' reflects the parade ground analogy upon which it is based; the user is the drill sergeant and the system the subservient private. The private speaks only when he is spoken to. When the drill sergeant speaks he supplies in a single command all the information required to carry out the task he wants accomplished. This involves identifying the task and possibly supplying any data values which the task requires. For example

'Halt' — task requiring no data values
or 'Peel me a grape' — task and associated data values

Traditional command language dialogues operate in teletype mode. The system says nothing except to display a constant prompt (such as the ubiquitous 'drive>' of most microcomputer operating systems) to indicate its readiness to obey. Each command is entered on a new line, and is usually terminated by a 'carriage return' or 'enter' character.

A>**dir**
A>**pip b:=a:*.com**
A>**mode com1:9600,e,7,1,p**

The private does not question an order; if told to do something stupid, he proceeds to try to do it. Command languages basically adopt the same approach and assume that the user knows what he is doing. There may not be many cases of soldiers marching off a cliff because the sergeant said 'right wheel' instead of 'left wheel' but there are many cases where, for example, a old version of a file has overwritten a new version because the user got the copy command wrong.

If a command is impossible to carry out, the private may say so without identifying the particular part which is impossible. Similarly, when further processing of the input line is impossible, a command language structure gives up, usually with a fairly non-specific error message, and the whole input must be repeated. For example,

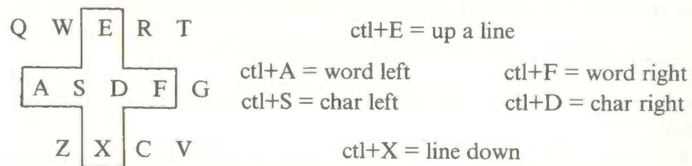
A>**dri a.*.com**
Bad command or filename (what is invalid?)

5.3.2 Design Criteria

Like the menu structure, a command language is appropriate for Input Control messages; it can, however, cater for a very wide range of options at any point in the dialogue, and does not require the background tasks to be hierarchically structured. Hence, it is appropriate to applications like operating systems where the background tasks form a flat structure of equal but distinct tasks, a sort of primaeval soup in which the tasks float like croutons waiting to be speared with a command fork!

Although it can support a relatively large set of commands, in practice the number in common usage is normally limited to reduce the load on a user's memory. Command language is the least supportive structure, and is appropriate to experienced and frequent users. Initial training is necessary before a user can use the system and he will only discover the full range of the system's facilities by external instruction rather than by using the system itself. Furthermore, since the system has no way of knowing what the user wishes to do, it is difficult to provide help facilities other than of a very general nature.

Because of the load imposed on a user's memory by the structure, it is important that the command identifiers are chosen so as to have a commonsense interpretation and to be easy to remember. Naming of the commands, or even positioning of command keys, can provide perceptual clues to aid memory. Thus the cursor positioning commands in WordStar reflect their layout on the keyboard.



This naturalness has not always been obvious in the command dialogues which have been implemented; it is doubtful whether many people who did not work with DEC equipment in the 1960s would immediately associate the PIP (standing for Peripheral Interchange Program) command of CP/M with a copy operation. The UNIX operating system provides some even more exotic examples.

The designer must guard against *excess functionality* resulting from an attempt to cater for every possible combination of task requirements with a *single command line*; that is, against developing a multiplicity of different commands, often performing overlapping functions. Such attempts at 'helpfulness' often result in a bewildering array of command keywords and syntaxes, most of which are seldom used and which confuse the majority of users.

The dialogue must handle data messages. This is typically done in a command language structure by means of compound input messages where the *command keyword* (input control) is followed by a *parameter list* (input data). For example:

PIP newfile=oldfile

contains the command keyword 'PIP' identifying a copy task and the parameter list 'newfile=oldfile' specifying that the contents of the file named 'oldfile' are to overwrite the contents of a file named 'newfile', or to create 'newfile' if it does not already exist. A parameter list may be expressed in one of two formats: positional parameters or parameter keywords.

The meaning of a *positional parameter* value is defined by the relative position it occupies in the command string. Thus, in the example

COPY thisfile newfile

the first parameter is the 'source file' (the file to be copied), and the second parameter is the 'destination file' (the new file to be created). A delimiter such as a comma, a slash or a number of spaces is used to separate one parameter from another.

With *parameter keywords*, each parameter value is preceded by a predefined identifier which specifies its meaning. Thus, in

COPY SOURCE=thisfile DESTINATION=newfile

the kewords SOURCE and DESTINATION identify which filename is which.

Positional parameters reduce the volume of input, but it is obviously essential that the values are entered in their correct order. Since rather unpleasant results can arise if you get the source file and the destination files the wrong way round, it is unfortunate that two widely used operating systems (CP/M and MS-DOS) use different orders in their respective copy commands! Positional parameters become particularly trying when the parameter list is long; some operating system commands take a dozen or

more parameters. When parameters can be omitted by entering two separators together at the appropriate position, this complexity is compounded.

Parameter keywords reduce memory load in one respect since order is no longer significant, and optional parameters can simply be omitted; it introduces another load, however, by requiring the user to remember more keywords, and the designer to invent 'meaningful' names for them. This approach also requires more processing by the system to cater for the recognition of keywords, and the flexible order.

In many command languages, the parameter list may also contain *switches* which alter the way in which the command is interpreted. Switches normally may occur anywhere in the parameter list and are denoted by an identifier; the prefix '—' is used in UNIX and the prefix '/' in MS-DOS. Thus, in UNIX (MS-DOS), a short form directory listing is invoked by the command

ls mydirectory (dir mydirectory /w)

and an extended directory listing with file sizes and dates by

ls -l mydirectory (dir mydirectory)

Many command languages support *macros* as a means of providing increased functionality in a single input without increasing the number of commands. A macro consists of a series of separate command strings stored as individual lines in a text file, called a Submit file in CP/M, a Batch file in MS-DOS and a shell script in UNIX. When the file name is entered, the individual command strings of the macro are executed one after another, as though they had been typed at the keyboard. Symbolic parameters may be specified in the command lines of the macro; these are replaced by the actual values entered as parameters of the macro when it is invoked. Thus if an MS-DOS macro called CLG.BAT contains the lines:

```
pas %1.pas
link %1.obj
%1
```

entering the command line 'clg myprog', will cause the execution of

```
pas myprog.pas
link myprog.obj
myprog
```

5.3.3 Implementation

There is an obvious similarity between a command input and form input. A Command input can be considered as supplying a series of answers to a series of implicit questions. Thus the input

A>**copy FromFilename ToFilename**

could be considered as answers to the implicit questions

command : **copy**
source : **FromFileName**
destination : **ToFileName**

Like the form, there must be some criteria for deciding that the user has completed the input; for a command, CarriageReturn acts as the terminator. However, unlike the form, the set of questions is not known in advance. It is not known until the particular command has been identified or, in some cases, which variant of the particular command. For example, the MS-DOS mode command comes in three different flavours:

mode {integer} to set the screen mode
mode com1:{BaudRate},{Parity},{DataBits},{StopBits},{Timeout}
to configure the serial port characteristics
mode lpt1:={Physical}
to assign a physical port to the screen echo

The input process for a command language is usually compared with the parsing of a program statement by a compiler or (more accurately) an interpreter. Although the basic syntax of an input is very rigid, there may be several variants of a command and there is often a good deal of freedom about things like how many, if any, spaces may appear in it. The software to implement it is considerably more complex than that required for the other structures. Instead of matching a single response, the dialogue process must first split the command into its constituent parts (called *tokens*), and carry out a greater number of matching operations to determine what option is required, and what data values are being passed to which parameters.

Figure 5.9 shows possible pseudo-code to implement a Command Language structure. GetResponse gets a string of characters from the keyboard, ending with a terminator character. The ReadField process could be used to accomplish this. GetToken gets the next token from 'Response'. If GetToken is applied four times to a Response of 'PIP newfile=oldfile', it will successively return the tokens 'PIP', 'newfile', '=', and 'oldfile'.

```
repeat
  Display BarePrompt
  GetResponse
  Matched:=true
  Set ValidTokens to possible commands
  while Matched and MoreInput
    GetToken
    Match Token against ValidTokens
    if not Matched
      Display ErrorMessage
    else
      Set ValidTokens to permitted values of next parameter
    endif
  endwhile
  if Matched check if Complete
until Matched and Complete
```

Fig. 5.9. Pseudo-code for a command language structure.

5.3.4 Summary

A command language is potentially the quickest and most flexible of all structures, and the majority of 'natural language' dialogues are basically command language structures with a very extensive vocabulary. Experienced users enjoy the feeling of controlling the system rather than being controlled by it. However, it offers little support and even experienced users find it difficult to utilise the full power; most are familiar only with the very limited subset of facilities which they use regularly.

In fact many of the desirable aspects of a command language may be mimicked with the Q&A structures and processes discussed in the preceding sections. We will return to this topic in Chapter 9.

5.4 Hybrid Dialogues

It should now be apparent that the four basic dialogue structures are not totally distinct but are in fact all variations of the Question and Answer structure.

A menu structure is Question and Answer modified so that a first level help

message, the menu, is displayed automatically before the option selection question is asked. A form filling structure displays a sequence of questions, the form, all at once, then asks for answers one at one time. A command language structure, particularly where positional parameters are used, is Question and Answer in which the user makes extensive use of 'answer ahead', i.e. supplies the answers to a series of implicit questions in response to a standard first question, the command prompt. These structures might be referred to as:

Menu	:	help-ahead
Form	:	question-ahead
Command	:	answer-ahead

Recognition of this fact provides some clues as to how a computer dialogue might adapt to different environments; we expand on this in Chapter 9. It also provides reasonable guidelines as to where each version of the Q&A structure might best be used.

A menu structure will be appropriate in cases where:

- the range of possible inputs is sufficiently small that they can be explicitly displayed;
- the user needs the possible inputs to be displayed.

This suggests that a menu should be used where a user who is inexperienced, or who is mainly using a pointing technique, is choosing from a limited range of values, i.e. typically in selecting a task process.

A command language structure will be appropriate in cases where:

- the number of input values is small enough to be remembered;
- a limited number of responses is sufficient both to identify the task required and to supply its data.

This suggests usage by experienced users where there is a fairly flat hierarchy of task processes with limited data input requirements.

A Form structure will be appropriate in cases where:

a standard sequence of inputs can be predicted

suggesting usage for the entry of transaction data.

The basic Question and Answer structure is a reasonable compromise for various levels of user. It can be used to substitute for all of the above, but will be particularly appropriate where:

- the range of input values is too great for a menu structure or too complex for a command language;
- the next question to be asked depends on the reply to the current question.

An obvious problem with the simple classification above is that, even with a given level of user familiarity, the data requirements in different parts of the system will vary. If you examine the dialogue requirements of a simple ledger system it will have aspects which are concerned with

- selecting a task process from a small set ot options, e.g. update the ledger, print reports, maintain customer codes;
- the input of transactions, e.g. entering invoice details;
- answers to an unpredictable sequence of questions with a wide range of possible values, e.g. in specifying selection criteria for reports.

Consequently, whilst most systems have a basic underlying Q&A, menu or command structure, it is rarely possible to produce a dialogue for a complete system using a single structure. Different parts of the dialogue will require different structures depending on their particular characteristics. In other words, most dialogues will represent a *hybrid* comprising several of the basic structures.

5.5 The Spreadsheet

Two applications, the spreadsheet and word processing, reflect this hybrid nature very well. We concentrate here on a typical spreadsheet dialogue, using the popular package Lotus 1-2-3 as an example.

The term spreadsheet derives from the large sheets of paper, ruled into lines and columns, which are used as working documents by accountants. The user is not restricted to fixed screen-sized chunks of 24 × 80 character positions but can position a screen sized 'window' anywhere over a much larger area. In Lotus 1-2-3, the spreadsheet can have a maximum area of 2048 lines by 256 columns, with each column consisting of several character positions. Both rows and columns are labelled:

rows : 1 — 2048
columns : A,B,. . . . . . .,Z,AA,AB,. . . . .,IV

A *cell* is identified by a column and row identifier (for example GF17). Each cell can hold a number, or a string of characters. The width of the cells

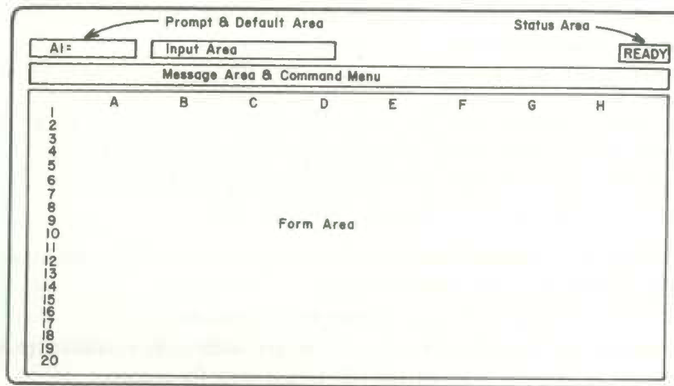


Fig. 5.10.

```
C14: +C8+C9+C10+C11+C12                                                READ

      A          B          C         D         E         F         G         H
1                          1985      1986      1987      1988      1989      1990
2
3    Sales               1126.77
4    Cost of Sales        300.77
5    ---------
6    Gross Profit         826.00
7    ---------
8    Warehousing           15.29
9    Distribution          20.70
10   Selling               31.45
11   Advertising           48.76
12   Administration        25.98
13   ---------
14   Operating Expenses   142.18
15   ---------
16   Operating Profit
17   Financing Costs
18   ---------
19   Profit before Tax
20   ---------
```

Fig. 5.11.

can be varied. When a spreadsheet is invoked the screen is divided into several distinct areas, as illustrated in Fig. 5.10. In the top left-hand corner of the screen is a prompt requesting the contents of the current cell and a prompt of this type is preserved through the interaction, a basic Question and Answer structure. In response to this prompt the user may enter:

- a numeric value in various formats;
- a text string;

- a formula such as +B1+ABS(B2) meaning the sum of B1 and the absolute value of B2.

In Fig. 5.11, the formula +C8+C9+C10+C11+C12 has been entered for Cell C14. The computer calculates 15.29+20.70+31.45+48.76+25.98 (the contents of cells C8..C12 respectively), and enters the answer 142.18 in C14. If we were to change the value of C9, the system remembers that C14 is dependent on C9 and changes its value accordingly.

The user may override the prompt with a command by entering a '/' character which will cause a menu bar of possible commands to appear, as shown in Fig. 5.12. Selection from the menu can be accomplished by scrolling through the menu with the cursor control keys and selecting with the return key; the current option is highlighted in inverse video and a description of its features appears on the line below. The user may also select by keying in the unique first character of the option name as an identifier.

Some of the commands require the user to specify a range of cells to which the command is to be applied. For example you may specify an output format such as 'currency' (two decimal places) which is to apply to a group of cells. This can be done either by listing the range of cells in answer to the prompt, or by 'painting' the area with the cursor control keys, i.e. by direct pointing (Fig. 5.13).

There is also a pure command language structure using function keys. You can skip to a particular cell on the spreadsheet directly by depressing function

```
C3:  (F2) 1126.77                                                  MENU
Worksheet  Range  Copy  Move  File  Print  Graph  Data  Quit
Format, Label-Prefix, Erase, Name, Justify, Protect, Unprotect, Input

        A         B         C         D         E         F         G         H
1                        1985      1986      1987      1988      1989      1990
2
3   Sales             1126.77   1183.11   1242.27   1304.38   1369.60   1438.08
4   Cost of Sales      300.77    315.81    331.60    348.18    365.59    383.87
5   ---------------------------------------------------------------------------
6   Gross Profit       826.00    867.30    910.67    956.20   1004.01   1054.21
7   ---------------------------------------------------------------------------
8   Warehousing         15.29     15.29     15.29     15.29     15.29     15.29
9   Distribution        20.70     20.70     20.70     20.70     20.70     20.70
10  Selling             31.45     31.45     31.45     31.45     31.45     31.45
11  Advertising         48.76     48.76     48.76     48.76     48.76     48.76
12  Administration      25.98     25.98     25.98     25.98     25.98     25.98
13  ---------------------------------------------------------------------------
14  Operating Expenses 142.18    142.18    142.18    142.18    142.18    142.18
15  ---------------------------------------------------------------------------
16  Operating Profit   683.82    725.12    768.49    814.02    861.83    912.03
17  Financing Costs    457.90    457.90    457.90    457.90    457.90    457.90
18  ---------------------------------------------------------------------------
19  Profit before Tax  225.92    267.22    310.59    356.12    403.93    454.13
20  ---------------------------------------------------------------------------
```

Fig. 5.12.

H19: (F2) +H16-H17 POINT
Enter range to format: C3..H19

	A B	C	D	E	F	G	H
1		1985	1986	1987	1988	1989	1990
2							
3	Sales	1126.77	1183.11	1242.27	1304.38	1369.60	1438.08
4	Cost of Sales	300.77	315.81	331.60	348.18	365.59	383.87
5	----------------	--------	--------	--------	--------	--------	--------
6	Gross Profit	826.00	867.30	910.67	956.20	1004.01	1054.21
7	----------------	--------	--------	--------	--------	--------	--------
8	Warehousing	15.29	15.29	15.29	15.29	15.29	15.29
9	Distribution	20.70	20.70	20.70	20.70	20.70	20.70
10	Selling	31.45	31.45	31.45	31.45	31.45	31.45
11	Advertising	48.76	48.76	48.76	48.76	48.76	48.76
12	Administration	25.98	25.98	25.98	25.98	25.98	25.98
13	----------------	--------	--------	--------	--------	--------	--------
14	Operating Expenses	142.18	142.18	142.18	142.18	142.18	142.18
15	----------------	--------	--------	--------	--------	--------	--------
16	Operating Profit	683.82	725.12	768.49	814.02	861.83	912.03
17	Financing Costs	457.90	457.90	457.90	457.90	457.90	457.90
18	----------------	--------	--------	--------	--------	--------	--------
19	Profit before Tax	225.92	267.22	310.59	356.12	403.93	454.13
20	----------------	--------	--------	--------	--------	--------	--------

Fig. 5.13.

C3: (F2) 1126.77 EDIT
Enter address to go to: D12

	A B	C	D	E	F	G	H
1		1985	1986	1987	1988	1989	1990
2							
3	Sales	1126.77	1183.11	1242.27	1304.38	1369.60	1438.08
4	Cost of Sales	300.77	315.81	331.60	348.18	365.59	383.87
5	----------------	--------	--------	--------	--------	--------	--------
6	Gross Profit	826.00	867.30	910.67	956.20	1004.01	1054.21
7	----------------	--------	--------	--------	--------	--------	--------
8	Warehousing	15.29	15.29	15.29	15.29	15.29	15.29
9	Distribution	20.70	20.70	20.70	20.70	20.70	20.70
10	Selling	31.45	31.45	31.45	31.45	31.45	31.45
11	Advertising	48.76	48.76	48.76	48.76	48.76	48.76
12	Administration	25.98	25.98	25.98	25.98	25.98	25.98
13	----------------	--------	--------	--------	--------	--------	--------
14	Operating Expenses	142.18	142.18	142.18	142.18	142.18	142.18
15	----------------	--------	--------	--------	--------	--------	--------
16	Operating Profit	683.82	725.12	768.49	814.02	861.83	912.03
17	Financing Costs	457.90	457.90	457.90	457.90	457.90	457.90
18	----------------	--------	--------	--------	--------	--------	--------
19	Profit before Tax	225.92	267.22	310.59	356.12	403.93	454.13
20	----------------	--------	--------	--------	--------	--------	--------

Fig. 5.14.

key F5 followed by the identifier of the target cell. If necessary, the 'window' is moved so that it now includes this cell (Fig. 5.14).

Lotus 1-2-3 was developed in the late 1970s and its dialogue structure is now considered a little old-fashioned; also, a consequence of trying to display as much of the spreadsheet data as possible on a single screen tends to be a somewhat cluttered layout. However, such packages illustrate many principles of good dialogue design. The spreadsheet approach is a *natural* mechanism for the types of application (such as financial analysis) at which it

is aimed because it is familiar to its prospective users. Spreadsheets typically form an integrated suite of packages with simple business graphics and file handling, all using a *consistent* approach and with data easily transferred from one to the other. Different dialogue structures have been built around the basic Question and Answer structure to reflect differing input data requirements, and those features of the hardware for which the package is intended are utilised to *minimise* the input effort. There is some *flexibility* in the structure to cater for differing user levels so that where an inexperienced user may well use scroll menus, the experienced user can execute his desired action directly by entering a command string made up of the option identifier. There is no reason why similar features should not be provided in any computer system.

5.6 Implementing a Hybrid Structure

The implementation of a structure like a spreadsheet should not appear too daunting in light of the dialogue processes which we have developed.

Each cell is a field and the spreadsheet is an array of fields, as in a Form. There is an obvious complication in that the number of cells is, in general, too large to display on a single screen; we will consider how this can be overcome in Chapter 10. There is also the complication that a cell may contain, as in the example of Fig. 5.10, not the actual content but an expression specifying how the content may be calculated. If, however, we drop these two requirements, the processes can provide the facilities required.

There is a QandA_DIB for the basic Q&A structure which requests the content of the current cell. As the user moves around the spreadsheet, the content of the question will change to contain the identifier of the current cell:

ChangeFieldContent(QandA_DIB.question,CurrentCellID);

The input will be treated as an arbitrary data value to be validated against a set of templates:

a numeric value
a text string
a '/' indicating that the command menu is to be displayed.

There will be a series of MenuDIBs and associated QandA_DIBs to represent the bar menus. The processes which we have defined so far allow the user to select from these menus either by scrolling or by keying an identifier.

However, this selection mechanism is predefined by the SelectionBy identifier in the QandA_DIB, whereas the example of Fig. 5.11 allows the user to do either. In other words, we need

SelectionBy = (id,scroll,position,key)

where 'key' indicates the input via the keyboard of either an identifier or a scrolling control.

The facilities illustrated in Figs. 5.13 and 5.14 require a slight extension. In the ControlDIB, we defined a number of standard control values which may be input in response to any question; these values include the abort and help request characters and the accept and pointing controls. In the spreadsheet there are additional standard responses, such as the function key commands; these differ from the other controls in that they are specific to a particular application rather than applicable to any dialogue. We can cater for this facility by extending ControlDIB to include these application control values. This is the final element which was reserved in the previous definition of ControlDIB. The generalised ControlDIB now becomes as illustrated in Fig. 5.15.

Whenever an input process receives a control value, it terminates immediately and returns the value to the dialogue via the ControlBuffer of ControlDIB so that it can take whatever action is necessary. These control values are defined by the control set, comprising

ControlSet:=Application Control +
[RequestAbort,RequestHelp,
PointNextTarget,PointPriorTarget,AcceptTarget,
PointNextField,PointPriorField,AcceptForm]

```
ControlDIBtype = record
              ControlBuffer      : byte;
              rubout             : byte;
              EchoSwitch         : OffOn;
              AcceptField        : byte;
              RequestAbort       : byte;
              RequestHelp        : byte;
              PointNextTarget    : byte;
              PointPriorTarget   : byte;
              AcceptTarget       : byte;
              PointNextField     : byte;
              PointPriorField    : byte;
              AcceptForm         : byte;
              ApplicationControl : SetOfByte;
              end;
```

Fig. 5.15. The general ControlDIB.

```
procedure ReadField(var field:FieldType;
                    DataSet:SetOfByte;
                    var ControlDIB:ControlDIBtype);
var key         : byte;
    ControlSet  : SetOfByte;
    EditSet     : SetOfByte;
    filter      : SetOfByte;
    complete    : boolean;
begin
with ControlDIB,field do
   begin
   DisplayField(field);
   ControlSet:=ApplicationControl+
               [RequestAbort,RequestHelp,
                PointNextTarget,PointPriorTarget,AcceptTarget,
                PointNextField,PointPriorField,AcceptForm];
   if slot.width=0 then
      {control characters only}
      ControlBuffer:=GetFilterKey(wait,NoEcho,ControlSet)
   else
      {both control and data}
      begin
      ControlSet:=ControlSet+[AcceptField];
      EditSet:=[rubout];
      filter:=ControlSet+DataSet;                      {nothing to edit}
      if attributes.justification=right then
         CursorTo(slot.row,slot.col+slot.width-1)
      else
         CursorTo(slot.row,slot.col);
      complete:=false;
      key:=GetFilterKey(wait,NoEcho,filter);
      if key in ControlSet then complete:=true
                           else content:='';           {clear answer field}
      while (not complete) do
         begin
         if key in EditSet then delete(content,length(content),1)
                           else content:=concat(content,chr(key));
         if EchoSwitch=on then DisplayField(field);
         if (length(content)=slot.width) and (AcceptField=0) then
            complete:=true
         else
            begin
            CursorTo(slot.row,JustifiedCol(field));
            if length(content)=0 then
               filter:=ControlSet+DataSet              {nothing to edit}
            else if length(content)=slot.width then
               filter:=ControlSet+EditSet              {ignore excess data}
            else
               filter:=ControlSet+DataSet+EditSet;     {allow any}
            key:=GetFilterKey(wait,NoEcho,filter);
            if key in ControlSet then complete:=true;
            end;
         end;
      if key=AcceptField then
         ControlBuffer:=0        {because it has been actioned}
      else
      if key in ControlSet then
         ControlBuffer:=key
      else
         ControlBuffer:=0;       {slot full and AcceptField=0}
      end;
   end;
end; {ReadField}
```

Fig. 5.16. A generalised keyboard input process.

Thus the generalised keyboard input process, ReadField, introduced in Chapter 3 can be implemented by the procedure of Fig. 5.16. This procedure repeatedly takes characters from the keyboard until

either a key representing a value in ControlSet is entered, or
if no AcceptField terminator has been specified, the result field content has been filled.

Data characters specified in the DataSet filter are stored in the content of the result field and, if echoing is specified, are displayed according to the slot and attributes defined by the result field. The content of the result field is edited with the 'rubout' key. On completion of ReadField, the result field holds any text characters which have been entered, and the keycode of the control character which caused termination is returned in ControlBuffer.

The function JustifiedCol determines the required cursor position based on the field justification and the number of characters already entered. Appendix G contains details of a suitable function.

Since ReadField can cater either for the input of a text string or the input of control characters we can utilise it to handle selection either by an identifier or by scrolling within a procedure (described in Appendix H) of the form:

```
procedure ChooseByKey(var QandA_DIB:QandA_DIBtype;
                      var ControlDIB:ControlDIBtype);
```

In order to implement selection from the bar menu of Fig. 5.12, we must define a ControlDIB which contains control values both for keyed identifiers and for scrolling, and an appropriate QandA_DIB:

```
with ControlDIB do
   begin
   rubout:=0;                           {no editing}
   EchoSwitch:=off;
   AcceptField:=CR;                     {we require a Carriage Return}
   PointNextTarget:=CursorRight;        {keycode 205}
   PointPriorTarget:=CursorLeft;        {keycode 203}
   AcceptTarget:=CR;                    {keycode 13}
   end;
with QandA_DIB do
   begin
   CreateField(question,'',0,0,0,''); {a null question}
   CreateField(answer,' ',0,0,1,'');  {a 1 character input field}
   InvertField(answer);    {defining target highlight as inverse}
   filter:=[ord('w'),ord('r'),ord('c'),... ord('q')];
   ErrorFlag:=off;
   CreateField(ErrorMessage,'',0,0,0,''); {no error message}
   NumberOfTargets:=9;
   CreateField(TargetList[1],'w',2,1,1,'');
   CreateField(TargetList[2],'r',2,12,1,'');
   CreateField(TargetList[3],'c',2,19,1,'');
   .........................................
   CreateField(TargetList[9],'q',2,57,1,'');
   SelectionBy:=key;
   CurrentTarget:=1;
   end;
```

Processing of the input can then be accomplished with the statement:

ChooseByKey(QandA_DIB,ControlDIB);

which will return the ordinal of the option chosen in CurrentTarget and its target name in answer.content, regardless of whether it was chosen by scrolling or by keying the identifier.

5.7 Mode and Modeless Operation

In some hybrid dialogues, the same input message may be interpreted differently according to an internal setting within the dialogue. Such dialogues have different *modes* of operation; the mode determines the context in which the input is interpreted. A *modeless* dialogue is one in which any given input will always be interpreted in the same way.

Consider a word processing package. A user who is editing a piece of text may wish a phrase to be inserted at a particular point, or he may wish the same phrase to replace an existing phrase located at that point. Common implementations of this define two modes of operation for the editor: an insert mode and a replace mode. In the former, any character entered at the keyboard will be interpreted as a request to insert that character immediately prior to the current cursor position; in the latter, the same character will be interpreted as a request to overwrite the current cursor position with that character.

Many packages take this concept of modes much further. A word processor may have two operating modes, edit mode or command mode; a user can usually change from one mode to another by entering a special control character. In edit mode, an input string 'smyfile' will be interpreted as a string of characters to be included in the text at the current position. In command mode, it might be interpreted as a request to quit the word processor and save the current text in a file called 'myfile'.

The obvious advantage of having different modes of operation is that natural identifiers can be used to invoke special control actions. For example, a command can be invoked by an easily memorised keyword like 'save' instead of an obscure, but unique, key sequence or function key; the mode provides a context which renders uniqueness unnecessary. However, it does not require much imagination to appreciate the dangers of this approach. The user must always ensure that he is in the correct mode for the operation he wishes to carry out. It is not unknown for users of such packages

inadvertently to delete a file by entering an 'unfortunate' text string whilst in command mode; the number of users who failed to save a file by inserting a string of the form 'shisfile' into the text of the file is legion!

Not surprisingly, all experimental evidence confirms that modeless operation is much to be preferred. Where there is no satisfactory alternative to different dialogue modes (the insert/replace option in text editing is a typical example) the current mode should always be clearly indicated in a status display.

5.8 Input Events — Handling Input from Several Input Processors

So far we have only considered cases where the dialogue process knows at any point from which processor the input will arise. The answer to a question will either come from the keyboard via one of the procedures ChooseById, ChooseByScroll or ChooseByKey, or it will come from a separate pointing device via the procedure ChooseByPosition. Suppose, however, that the user may select an answer using direct pointing or may request help by pressing a function key on the keyboard. Which input driver process should the dialogue process call?

In such a case, there are a number of *input events* (actions on some input processor) which can occur. The dialogue process needs to know when an event has occurred and what type of event it is. We will consider the case where the user's response to a 'question' may come either from the keyboard or from a two-button mouse operating in real mouse mode like that described in Chapter 3; the method is readily extensible to other devices. We assume the existence of the following additional functions in the keyboard and mouse driver processes:

```
function KeyboardEvent:boolean;external;
function PointerEvent:boolean;external;
```

These return true if an activity has occurred on the device and false otherwise. The next question to consider is what is meant by an 'activity' or event on the device.

An event might be triggered on the keyboard by the depression of any key. As we saw in Chapter 3, this will typically cause a keycode value to be stored in the keyboard buffer; thus, a keyboard event has occurred if and only if there is something in the keyboard buffer. With most operating systems, low

SKYHAWKE Ex. 1016, page 68

level routines can be provided to inspect this buffer and determine whether it is empty or not. In fact, many operating systems permit a more selective definition of events. An event may be defined as the depression of one of a given subset of keys, i.e. there is a procedure

procedure DefineKeyboardEvent(filter:SetOfByte);external;

such that the function KeyboardEvent returns true only if the user has pressed a key which produces a keycode value in filter. Thus the filter should be defined as the set of keys to which the ReadField process is sensitive:

```
EventFilter:= QandADIB.filter +
              [rubout,AcceptField,RequestAbort,RequestHelp,
               PointNextTarget,PointPriorTarget,AcceptTarget,
               PointNextField,PointPriorField,AcceptForm] +
               ApplicationControl
```

i.e. {data characters for a keyed answer} + {ControlDIB input control values} + {any application-specific control values in ControlDIB}

An event might be triggered on the mouse by a movement in a given direction, or by the press or release of a particular button. Again, it is usually possible to define by software which activities or combination of activities trigger an event. For our purposes, it will be sufficient to define a pointer event as triggered by a movement in any direction or a press of any button.

We can now develop routines to test for an input event and return its value to the dialogue process. As with the keyboard processes described in Chapter 3, the dialogue process may wait for an input event to occur, or merely check whether one has occurred. Suitable functions are illustrated in Fig. 5.17. Note that the ordering of the conditional statements means that pointing events take priority over keyboard events; a keyboard event will be reported only after all outstanding pointer events have been reported.

The function, WaitInputEvent, can be used to develop a generalised dialogue process which caters for input from either the keyboard or a pointing device, and for selection by identifier, scrolling or pointing. This Choose process, illustrated in Fig. 5.18, consolidates all the procedures — Choose-ById, ChooseByScroll, ChooseByPosition and ChooseByKey — which we have developed so far.

The Choose procedure insists that the user select a valid target; it will terminate only when a non-zero CurrentTarget has been picked either implicitly or explicitly. With scrolling, only valid targets can be picked but a

```
type SelectionType  = (id,scroll,position,key,any);
     InputEventType = (null,keyboard,pointer)

function TestInputEvent(selector:SelectionType):InputEventType;
begin
if ((selector=any) or (selector=position)) and PointerEvent then
     TestInputEvent:=pointer
else
if (selector<>position) and KeyboardEvent then
     TestInputEvent:=keyboard
else TestInputEvent:=null;
end; {TestInputEvent}

function WaitInputEvent(selector:SelectionType):InputEventType;
var event : InputEventType;
begin
event:=null;
repeat
   event:=TestInputEvent(selector);
until event<>null;
WaitInputEvent:=event;
end; {WaitInputEvent}
```

Fig. 5.17. Checking for an input event.

user may key an unmatched identifier or may press a mouse button when not pointing at any target. Usually, the dialogue will require that the user select one of the targets; however, some systems which use direct pointing interpret the selection of no target as a request to exit. The minor changes necessary to support this facility are left to the reader.

As a final step, the generalised dialogue input process, Q&A, of Fig. 4.16 can be modified to incorporate the general selection process. This procedure, ilustrated in Fig. 5.19, provides a general input process which can be used with any of the Q&A or menu structured dialogues we have examined, or with any individual question and answer step within a form dialogue structure and will handle any type of input message from the keyboard or from a pointing device.

5.9 Summary

Both the form and the command language structure request a set of answers from the user.

The form structure displays a series of questions and requests the user to supply answers for each; most form dialogues permit the user considerable flexibility in choosing which question to answer next and in editing previous answers. It is a natural and supportive mechanism for the entry of transaction data regardless of whether or not it is based on an existing clerical form.

```
procedure Choose(var QandA_DIB:QandA_DIBtype;
                 var ControlDIB:ControlDIBtype);
var CursorWas   : OffOn;
    complete    : boolean;
    event       : InputEventType;
    PriorTarget : byte;
    row,col     : byte;
begin
with QandA_DIB,ControlDIB do
   begin
   CursorWas:=TestCursor;
   if (CurrentTarget>=1) and (CurrentTarget<=NumberOfTargets) then
      begin
      answer.content:=TargetList[CurrentTarget].content;
      HighlightField(TargetList[CurrentTarget],answer.attributes);
      end;
   if filter=[] then SwitchCursor(off);
   DisplayField(question);
   CursorAt(row,col);
   PointerTo(row,col);
   complete:=false;
   repeat
      if ErrorFlag=on then DisplayField(ErrorMessage);
      ErrorFlag:=off;
      if ControlBuffer=0 then {otherwise there is already something}
         begin
         event:=WaitInputEvent(SelectionBy);
         HideField(ErrorMessage);
         HideField(HelpMessage);
         if event=pointer then
            ReadPointer(row,col,ControlBuffer)
         else
            ReadField(answer,filter,ControlDIB);
         end;
      if equal(ControlBuffer,RequestAbort) then
         complete:=true
      else
         begin
         PriorTarget:=CurrentTarget;
         if event=pointer then
            {***** absolute pointing *****}
            begin
            CurrentTarget:=MatchPosition(row,col,TargetList,
                                         NumberOfTargets);
            CursorTo(row,col);
            if (CurrentTarget<>0) and (AcceptTarget=0) then
               complete:=true;
            end
         else {event=keyboard}
            begin
            if ControlBuffer=0 then
               {***** matching keyed data *****}
               begin
               CurrentTarget:=MatchString(answer.content,TargetList,
                                          NumberOfTargets);
               if CurrentTarget<>0 then complete:=true
                                    else ErrorFlag:=on;
               end
            else
            if (CurrentTarget<>0)
            and ((ControlBuffer=PointNextTarget)
            or  (ControlBuffer=PointPriorTarget)) then
               {***** relative pointing *****}
               begin
               if ControlBuffer=PointNextTarget then
```

Fig. 5.18. A generalised dialogue selection process.

```
              if PriorTarget=NumberOfTargets then CurrentTarget:=1
                              else CurrentTarget:=PriorTarget+1
           else
              if PriorTarget=1 then CurrentTarget:=NumberOfTargets
                           else CurrentTarget:=PriorTarget-1;
           ControlBuffer:=0;    {because it has been actioned}
           {do nothing in the case of AcceptTarget=0}
           end;
        end;
     if PriorTarget<>CurrentTarget then
        begin
        if PriorTarget<>0 then DisplayField(TargetList[PriorTarget]);
        if CurrentTarget<>0 then
           begin
              HighlightField(TargetList[CurrentTarget],
                             answer.attributes);
              CursorAt(row,col);
              PointerTo(row,col);
              end;
           end;
        if (not complete) and (ControlBuffer<>0) then
           begin
           if ControlBuffer=AcceptTarget then
              begin
              complete:=true;
              ControlBuffer:=0;    {because it has been actioned}
              end
           else
           if ControlBuffer=RequestHelp then
              begin
              DisplayField(HelpMessage);
              ControlBuffer:=0;    {because it has been actioned}
              end
           else
           if ControlBuffer in ApplicationControl+[PointNextField,
                                PointPriorField,AcceptForm] then
              complete:=true;
           end;
        if CurrentTarget<>0 then
           begin
           answer.content:=TargetList[CurrentTarget].content;
           DisplayField(answer);  {update the display on the screen}
           end
        else
           answer.content:='';
        end;
      if not complete then ControlBuffer:=0;
   until complete;
   if CurrentTarget<>0 then
      begin
      DisplayField(TargetList[CurrentTarget]);
      answer.content:=TargetList[CurrentTarget].content;
      end;
   if CursorWas=on then SwitchCursor(on);
   end;
end; {Choose}
```

Fig. 5.18. (*cont.*)

```
procedure QandA(var QandA_DIB:QandA_DIBtype;
                var ControlDIB:ControlDIBtype);
begin
if QandADIB.NumberOfTargets=0 then
   ArbitraryData(QandA_DIB,ControlDIB)
else
   case QandA_DIB.SelectionBy of
     id       : ChooseByID(QandA_DIB,ControlDIB);
     scroll   : ChooseByScroll(QandA_DIB,ControlDIB);
     position: ChooseByPosition(QandA_DIB,ControlDIB);
     key      : ChooseByKey(QandA_DIB,ControlDIB);
     any      : begin
               {define the input events for each device}
                Choose(QandA_DIB,ControlDIB);
                end;
     end; {case}
end; {QandA}
```

Fig. 5.19. A generalised dialogue input process.

The command language structure assumes an experienced user and requires, in response to a standard prompt, the input of a series of answers specifying both a task and its associated data values. The nature of data items required (the format of the command parameters) will typically not be known to the dialogue until after the task has been identified.

All four traditional dialogue structures are variants of the basic Question and Answer structure:

Menu	=	Q&A with help-ahead
Form	=	Q&A with question-ahead
Command	=	Q&A with answer-ahead

In general, no single structure can serve for the whole dialogue of a system; different areas of the system will have characteristics which suit different structures. Although most dialogues have an underlying structure which defines their style, they are usually hybrids consisting of a combination of the basic structures.

In modeless operation, an input will always be interpreted in the same way. Some dialogues support different modes of operation; each mode represents a different context and hence a potentially different interpretation of any input.

The commonality between the structures means that such hybrid structures can be implemented with abstractions based on two data structures:

- A ControlDIB specifying parameters which control the operation of the input process.
- A QandA_DIB specifying parameters which define the interaction for a particular question and response.

A handler for input events, which occur when an activity takes place on an input processor, enables the dialogue process to accept inputs from several input devices without knowing in advance which will provide the input.

This means that it is possible to develop a single dialogue process which can handle a variety of different input mechanisms in a variety of different dialogue structures, and from a variety of input devices. This process provides a strong base for ensuring both consistency and flexibility in a dialogue.

The dialogue structure is a major factor in determining its naturalness, consistency, non-redundancy, supportiveness and flexibility. However, these factors are influenced not only by the structure in which the system requests information from and presents information to a user, but also by the content of the messages exchanged. We consider this aspect in the next chapter.

Discussion Exercises

D1. In Chapter 2 you were asked to discuss the type of dialogue structure which is best suited the Mailsale application described in Appendix A. A form filling structure would be the normal approach. Should the form filling use immediate or deferred validation? Should it use manual skip or auto skip? Is it necessary for the clerk to be able to edit previous answers on the form?

D2. In a form filling dialogue, the questions on the form are predetermined and normally an answer is requested for each question. However, it is possible that the answer to a question will cause some later questions to be suppressed because they are now irrelevant. The forms procedures developed in the chapter do not cater for this; suggest what changes are necessary for it to be accommodated.

Programming Exercises

P1. Implement the library of Q&A routines and the library of MCI routines described in Appendices G and H.

P2. Develop a program which will accept the following data via a form, with immediate validation

```
Passenger Name  [                    ]
Destination     [            ]
Date            [  /  /  ]
Time            [  .  ]
```

Passenger name can be up to 20 alphabetic characters long. Destination can be any of Calais, Boulogne, Dieppe, Oustreham, Cherbourg or Le Havre; this field is selected via a suppressed menu. Date is in the format dd/mm/yy with today as the default. Time is in 24-hour format but leading zeros can be omitted.

P3. Develop a procedure to output the following boxed menu

dwarf: happy
happy dozy lazy sleepy grumpy

and allow the user to select from it either by keying the identifier of an option or by scrolling around the menu of options. If the user keys an identifier, it is echoed in the menu header; the default selection is 'happy'. The procedure will return the ordinal of the chosen dwarf.

P4. Develop a menu design aid program to assist in the design of scroll menus. The program will allow for a menu header field up to 30 characters in length and a maximum 10 menu items up to 15 characters long; all fields in the menu will have the same attributes.

The program will first request the user to specify these attributes. It will then draw a form which contains blank fields for the header and 10 menu items of maximum length; this will be centred in the middle of the screen. The user will overwrite these fields with the relevant contents; the size of the menu fields displayed will be amended to reflect the length of the contents.

The position of the header or the target items can be changed by pointing at the corresponding field and selecting it, moving the cursor to the desired position and selecting it again.

Further Reading

Gittings I. (1985) *Query Languages*, Edward Arnold.
Kraut R.E. *et al.* (1984) 'Command Use and Interface Design', Rosenberg J. 'A Featural Approach to Command Names', both in Janda A. (Ed) *Human Factors in Computing Systems*, North Holland.
Landauer T.K. *et al.* (1983) 'Natural Command Names and Initial Learning: A Study of Text Editing terms', *Comm.ACM*, **26**, 7.
Shneiderman B. (1986) *Designing the User Interface*, Addison Wesley.
Smith S. and Mosier J.L. (1984) *Design Guidelines for User Interface Software*, MITRE Corporation.
Zloof M.M. (1982) 'Office-by-example: A Business Language that Unifies Data and Word Processing and Electronic Mail', *IBM Systems J.*, **21**, 3.

Chapter 7

Screen formatting

7.1 Introduction

We saw in Chapter 4 that effective interaction between user and system requires a dialogue which is:

natural
consistent
non-redundant
supportive
flexible.

These criteria apply not only to the basic structure of the dialogue and to the content of the displayed messages, but equally to how they physically appear on the screen. The physical appearance of the screen depends on what message fields are displayed and on the choice of slot and of attributes for each of these.

The design process for any screen may be summarised as follows:

decide *what information*, i.e. what fields, will appear on the screen
decide *the basic format* of this information
decide *where it is to appear* on the screen, i.e. the slot for each field
decide *what highlighting is required*, i.e. what attributes are necessary for each field
develop a *draft screen* layout
evaluate the effectiveness of the layout

and repeat the process until both you and the prospective users are satisfied with the results.

We will illustrate this process by considering a design for a form filling input screen based on the document in Fig. 7.1. This covers most of the principles common to all screen design.

SKYHAWKE Ex. 1016, page 77

CAR FERRY RESERVATION FORM

OUTWARD VOYAGE		INWARD VOYAGE	RESERVED ACCOMMODATION				
First choice	From	From					
	To	To	Type of cabin preferred *delete as applicable	OUTWARD		INWARD	
Date				Night/Day*		Night/Day*	
Sailing Time			If whole cabin is not required, No. of Berths/Couchettes* *delete as applicable	Male	Female	Male	Female
Second choice	From	From					
	To	To					
Date			No. of Reclining Seats				
Sailing Time			No. of Club Class Seats				

NAME AND ADDRESS (Block capitals please)	VEHICLE DETAILS
Name	Reg. No.
Address (or Agent's stamp)	Overall length (inc. roof-top luggage) m — Height under 1.83m*/over 1.83m* (inc. roof-top luggage) *delete as applicable
	CARAVAN*/TRAILER* DETAILS *delete as applicable
	Overall length (inc. Tow Bar) m — Height under 1.83m*/over 1.83m* *delete as applicable
Post Code	Motorcycle Reg. No. — Solo/Combination* *delete as applicable
Telephone No.	PASSENGERS No. of Adults (inc. driver) — No. of Children (4 and under 14)
CHALET/CARAVAN/CAMPING SITE	INSURANCE
please tick appropriate box. ☐ Tent Rental ☐ Chalet ☐ Caravan/Camping Site	Holiday Insurance ☐ Vehicle Cover Extension* ☐ Caravan/Trailer Cover Extension† ☐
	Car make — Car model
	Date of Return if not stated above — Age of Vehicle if personalised number plate
	Please tick box if cover required for Winter Sports activities ☐

Fig. 7.1. Ferry reservation document.

7.2 General Guidelines for Screen Layouts

The clerical form illustrated in Fig. 7.1 is used for booking a ferry reservation. The booking clerk in a travel agency will either complete, or ask the customer to complete, a copy of the form. The clerk will then enter the relevant details, using a standard VDU and keyboard, into the ferry operator's computer booking system and wait for the booking to be confirmed. Although at first sight it may seem as though there is duplication of effort in completing the form rather than keying the customers answers directly into the system, it is often quicker and more accurate to do this where information such as name and address or registration numbers, which are susceptible to mishearing or misspelling, are involved.

```
CAR FERRY RESERVATION

               OUTWARD VOYAGE   INWARD VOYAGE     RESERVED ACCOMMODATION
1ST CHOICE FROM PRT TO DIP  FROM CHB TO WEY       CABIN  OUT DAY IN NIGHT
               1305 860506     2320 860521        BTHS/CHTS          1M 2F
2ND CHOICE FROM PRT TO DIP  FROM DIP TO PRT       RECLIN      0        0
CONFIRMED  1305  860506        0005 860520      CLUB CLASS SEATS  0       0
NAME AND  ADDRESS            VEHICLE DETAILS
NAME MR. A. BLENKINSOP       REGNO LAC939L
ADDRESS 47 ACACIA AVENUE        OVERALL LENGTH 3.5M HEIGHT N
           SUBURBIA             CARAVAN/TRAILER CARAVAN
           ANYTOWN              OVERALL LENGTH 3.0M HEIGHT Y
POSTCODE                        MOTORCYCLE REGNO          S/C
TELEPHONE NO:555123456          PASSENGERS: NO.ADULTS 4 NO.CHILDREN

CHALET/CARAVAN/CAMPING SITE     INSURANCE
TENT RENTAL N                   HOLIDAY   VEHICLE COVER  C/T EXTENSION
CHALET N                           N            Y               Y
CARAVAN/CAMPING SITE N          CAR MAKE VAUXHALL MODEL CAVALIER GLS
                                DATE OF RETURN          AGE
OK?                             WINTER SPORTS
```

Figure 7.2 A poor screen layout

Examine the screen layout in Fig. 7.2 which displays a completed input screen for a particular booking. Hopefully, most readers will feel that a number of improvements are possible in this layout which represents a slavish following of the actual clerical document in Fig. 7.1. Although existing clerical input and output documents provide many pointers to the desired format (the relative order of screen input and source document should be the same), they do not eliminate the need for a conscious process of screen design.

The layout should be such that the user can scan the screen in a logical order and can easily:

- extract the information which he is seeking;
- identify related groups of information;
- distinguish exceptional items (such as error or warning messages);
- determine what action (if any) is necessary on his part to continue with the task.

A user will find it very difficult to manage any of these with the layout of Fig. 7.2. We will attempt to produce a more satifactory layout by following the design process.

7.3 What Information Should be Displayed on the Screen?

Non-redundancy implies that the layout contains *only the information which is relevant* to the user at that point in the dialogue. The fact that other

information is available, or is stored together with the necessary information, or that there is room for more information on the screen is not significant. The user's need is the deciding factor and the designer must understand the user's task in sufficient detail to assess the information required to support it.

The layout in Fig. 7.2 contains superfluous information. The first and second choice voyages on the form in Fig. 7.1 are presumably intended for the case where the user is not present at the time the booking is made; there is no reason for displaying both on the output, when a particular booking has been made. The clerk will presumably input the first choice; if this is not accepted, then the first choice details are irrelevant and the second choice details can be entered in their place.

A similar argument applies to the layout of a menu. A menu should only list options, which the user can select, regardless of whether it is a traditional menu or a command bar of the type described for spreadsheets in Chapter 5. This is a common failing of many existing packages. In Prestel, a user is presented with a menu of all the possible options even though some choices are restricted to a subset of particular users, and many hybrid dialogues display the same command menu throughout, regardless of whether a given command is available at that point. In both cases, options which are unavailable may be indicated by some highlighting attribute.

Although the screen should not swamp the user with superfluous information, it is equally important to display *all the information relevant* to the user at that point. A user should not be expected to remember information from one screen so as to be able to process the information on a later screen. If all the items from a clerical document will not fit on a single screen, certain items may need to be repeated on all screens to preserve continuity. Thus, if a clerk is required to refer continually to a customer's name while undertaking a set of transactions which occupy several related screens, the customer name should appear in a consistent position on all the screens.

As well as deciding which individual items of information are required, the designer must consider how these items relate to each other. A *logical group*, a set of information which must be viewed as a composite entity in order to achieve the task purposes, should not be split across a screen boundary. Obviously, there can be no fixed rules for deciding what comprises a logical group; this will depend on the nature of the application. A number of data items may be logically related because they describe the same aspect of a task, because they occur from the same source, and so forth. A designer who understands the nature of the user's task and of the data which is involved should not find this a major problem. Very often, the format of existing

clerical inputs and outputs will provide strong clues to the groupings.

At first sight there appear to be six main logical groupings of the information in Fig. 7.1:

1. the voyage
2. reserved accommodation
3. vehicles and passengers
4. customer name and address
5. campsite reservation
6. insurances

The first two of these can be subdivided into an outward and an inward passage. The ferry company seems to assume that the same vehicles and number of passengers will be involved in each direction!

As a result of this process, the designer should be able to produce, for a particular task, a list of the required data items in their dependent groups, and to indicate whether these are optional(O) or mandatory(X). This is illustrated in Fig. 7.3, in which dependencies are denoted by indentation; thus registration number, length, height and caravan details are all dependent on a car being booked.

Thus the user must specify all the items in the 'Voyage Group'. No vehicles need be specified in a booking but there must be at least one adult passenger; there need be no child passengers. A car is specified by the input of a registration number; if no car is specified, there is no requirement for the input of the vehicle length and height, or for details of caravan or trailer. There is no need to distinguish between caravans and other types of trailer.

If accommodation can only be reserved on certain routes, there may seem little point in displaying these fields for routes on which it is inapplicable. We could choose to suppress the display of this information unless the user inputs a route on which it is applicable. However, accommodation may not be reserved even if it is available. As we saw in Chapter 5, this suppression or automatic skipping of input fields is not very compatible with the question-ahead nature of form filling and is desirable only if it reflects how the user would naturally scan the form. It may be better to provide a consistent input mechanism by always displaying these items.

There would seem to be little requirement for instructions on how to complete the form in our example. There will obviously be a requirement to confirm that the details are correct; it may be necessary to allow space for messages explaining how a user can edit the form if they are not.

Item				Status
Voyage			(1 group for Outward sailing and 1 for Inward)	
	From Port			X
	To Port			X
	On Date			X
	At Time			X
Accommodation			(1 group for Outward sailing and 1 for Inward)	
	Cabin			
		Type		O
		Berths/Couchettes — Male		O
		— Female		O
	Seats — Reclining			O
	— Club Class			O
Vehicles and Passengers				
	Car			
		Registration		O
		Length		O
		Over 1.83 m high		O
		Caravan		
			Length	O
			Over 1.83 m high	O
	Motorcycle			
		Registration		O
		Solo/Combination		O
	Passengers			
		No. of Adults		X
		No. of Children		O
Customer Name and Address				
	Name			X
	Address (× 3 lines)			X
	Post Code and Telephone Number			O
Campsite Reservation				O
Insurances				
	Holiday Cover (yes/no)			O
	Vehicle Cover (yes/no)			O
		Trailer Extension (yes/no)		O
		Car Make		O
		Car Model		O
		Date of Return		O
		Age		O
	Winter Sports			O

Fig. 7.3 Listing the items for the display.

7.4 How Should the Information be Displayed?

Having decided what items are necessary for the task, the next step is to decide the format in which they should appear. This provides the designer both with an indication of the slot size and of any special attributes which are desirable for each field.

Naturalness implies that the information is presented in *immediately usable* form. The user should not be required to manipulate the information, e.g. by looking up codes, or by computing row or column subtotals outside the system. Figure 7.2 does not confirm the input of a port code with a port name; this may be acceptable in this particular example, given the limited number of ferry ports. Dates and times should be formatted to normal conventions, not, as in Fig. 7.2, displayed in internal system representation. The designer should consider whether a textual presentation is immediately usable; for example, an impression of a sales trend is gained much better from a graph than from a table of figures.

The use of normal *upper and lower case* conventions improves the readability of text. Road signs display place names in both upper and lower case: the shape Birmingham conveys much more than the shape 'BIRMINGHAM'. The example in Fig. 7.2. was obviously produced by an old-fashioned programmer who even thinks in upper case! Accountants are accustomed to recognising as negative values which are enclosed in brackets or displayed in red, rather than preceded by a minus sign.

Caption fields identify the meaning of individual input and output data fields. They should clearly indicate the content of the corresponding data field and be well differentiated from the data values. This can be accomplished with the guidelines in Fig. 7.4.

Figure 7.2 contains a number of examples of the misuse of captions. Some data fields (notably the sailing times and dates) have no captions. Several seem needlessly verbose (CHALET/CARAVAN/CAMPING SITE) whilst others are arbitrarily abbreviated (RECLIN). Some captions fail to convey the meaning of the data field: HEIGHT YES is meaningless. Furthermore, it is almost impossible to distinguish between captions and the data values they are supposed to identify; captions appear alongside most data fields but above them in the Insurance section.

As a result of this process, the designer will be able to extend the list in Fig. 7.3 to include provisional lengths for the various input fields and provisional caption fields; this is illustrated in Fig. 7.5.

Caption names as brief as possible but without arbitrary abbreviation.

Captions distinguished from data fields by emphasising data values with any combination of

punctuation, decorators or case e.g.
Date: 6 May 86 or Date [6 May 86] or date 6 MAY 86

'weaker' attributes in the caption e.g.
Date **6 May 86**

Captions positioned in a natural and consistent physical relationship to the corresponding data field

on the same line and to the left for single occurrence fields, e.g.

Invoice Number: [**123456**]
Invoice Date : [**25/01/86**]

as column headings above the corresponding data field for multiple occurrence fields, e.g.

Part Number	Description	Stock Level
123456	**widget**	**750**
234567	**sprodget bucket**	**23**
376890	**something else again**	**7**

for heading information common to a number of items, such as a logically related group, centred above the group of fields

	Nominal Account Allocation	
Cost Centre	Account No.	Amount
123	**12345**	**140.79**
126	**12345**	**79.80**
999	**99000**	**32.17**

Fig. 7.4 Guidelines for captions.

SKYHAWKE Ex. 1016, page 84