Exhibit 1016 – Part 3

Group Caption Field	Size and Format	
1. Voyage Details	Outward	Inward
From	Code(3) Name(15)	Code(3) Name(15)
To	Code(3) Name(15)	Code(3) Name(15)
Date	(dd/mm/yy)	(dd/mm/yy)
Time	(hh.mm)	(hh.mm)
2. Accommodation		
Cabin	(D, N or null)	(D, N or null)
Berths — M	(nn or null)	(nn or null)
— F	(nn or null)	(nn or null)
Seats — R	(nn or null)	(nn or null)
— C	(nn or null)	(nn or null)
3. Vehicle Details		
Car Registration	(8 alphameric or null)	
Length	(99.9)	
Over1.83m	(y/n)	
Caravan Length	(99.9 or null)	
Over1.83m	(y/n)	
or		
M/C Registration	(8 alphameric or null)	
Solo/Combination	(s/c)	
4. Passenger Details		
Adults	(nn)	
Children	(nn or null)	
5. Customer Details		
Name	(20 alpha)	
Address (× 3 lines)	(3×20 alpha)	
Post Code	(8 alphameric or null)	
Telephone Number	(12 alphameric or null)	
6. Campsite Reservation	(T/C/S or null)	
7. Insurances		
Holiday	(y/n)	
Vehicle	(y/n or null)	
Trailer	(y/n)	
Car Make	(10 alpha)	
Model	(15 alpha)	
Return Date	(dd/mm/yy)	
Age of Vehicle	(nn years)	
Winter Sports	(y/n)	

Fig. 7.5 Adding the item formats.

7.5 Where should the Information be Displayed?

7.5.1 General Guidelines on Positioning

Having decided what items to display and in what format, the next stage is to position these into the available space on the physical screen. This is not just a case of squeezing them in wherever they will fit!

So much information is displayed in Fig. 7.2 that it is difficult to identify individual items or groups of items. *Clutter* is a subjective measure; whether a particular layout seems cluttered will depend on the individual user and the task which the screen is designed to assist. For example, a screen used for data entry from source documents by trained key operators can be more crowded than one designed for the display of output data messages or for input by an untrained operator. However, there are a number of rules of thumb concerning spacing which provide guidance:

leave approximately half of the total screen area blank;
leave a blank line after every fifth row of a tabular format;
leave four or five spaces between the columns of a columnar format.

These rules of thumb are precisely that, they should be used merely as guidelines, not followed slavishly. Whilst it is quite possible to produce an acceptable format in particular cases which does not follow them, a conscious design decision should have been taken to contradict them.

Where items must be split across more than one screen, the split should occur at a natural break, if such exists. Not only should the break not occur within a logical group, all groups required for a particular decision should also appear on a single screen. Consider what happens if a ferry booking is not accepted. This may occur either because the desired accommodation is not available or because there is insufficient vehicle space; it can be assumed that there is always sufficient passenger space. The customer may change his voyage or may forgo the desired accommodation or both. Thus the clerk requires voyage, accommodation and vehicle details to be displayed on a single screen. It is not essential that the other groups appear on this screen since they do not affect the success of the booking. This analysis also suggests seven logical groups as indicated in Fig. 7.5 rather than the six groups of Fig. 7.3.

On any screen, logical groups should be clearly identifiable as separate entities. This can be achieved by leaving several spaces around the borders of

Voyage Details	Outward	Inward
From	**dve** Dover East	**ost** Ostende
To	**blg** Boulogne	**flk** Folkestone
Date	**26.10.86**	**31.10.86**
Time	**14.05**	**17.50**

Fig. 7.6 Using 'boxing' to delimit logical groups.

each group or by explicitly 'boxing' with vertical and horizontal line segments (as in Fig. 7.6) with different attributes for fields in different groups.

The users's eye should be guided through the screen by the physical patterns created by the blocks of text. These blocks should be *balanced*. Fields should not be tucked up against the margins of the screen but centred about the vertical and horizontal axes. In cases such as menu screens where only a relatively small amount of information is to be displayed, it should appear centred in the leftmost two thirds of the screen. To reinforce this symmetry, data fields and captions should be aligned vertically within a logical group; where possible, this alignment should be preserved across all logical groups.

There should be an obvious *starting point*. The normal convention is to start at the top left hand corner and proceed left to right and top to bottom. Although boxing can suggest other conventions, it should only be changed as the result of a conscious design decision. The same type of information should appear in a *consistent and predictable relative position* on the screen throughout the application.

Aesthetics are important. A screen that is attractively presented is likely to invoke a positive response from a user and be easier to follow. You cannot reasonably expect a user to take more trouble completing a form on the screen than the designer took developing it!

7.5.2 A Template for Screen Layouts

If screen layouts are to be consistent both within and across applications, they must all be based on a common template. Figure 7.7 illustrates such a template.

The top two or three lines of the screen are reserved for *title and status information*. Titling information often includes a description of the

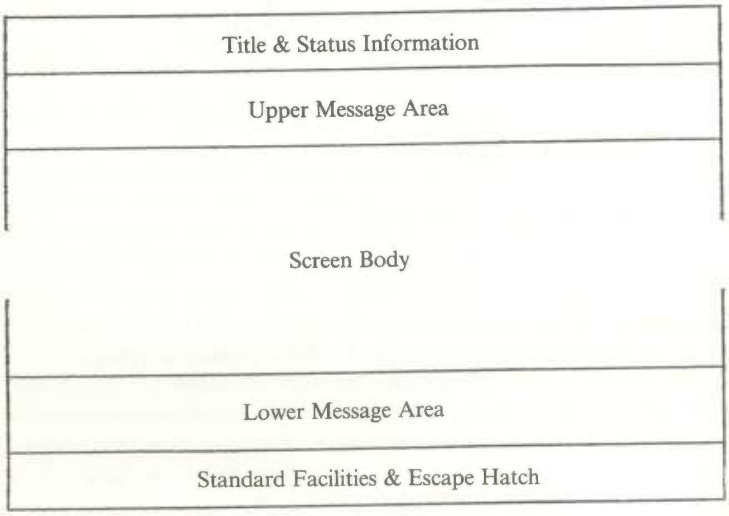


Fig. 7.7 A screen template.

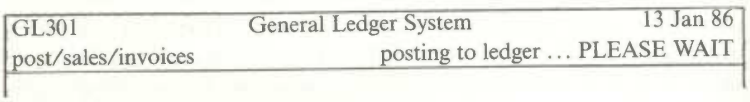


Fig. 7.8 Where am I? Is it still working?

application and/or particular task within it to which the screen relates. The current date and time may be displayed towards the right-hand margin. It is also common for each screen to be assigned a unique reference number; if a user encounters any problems with the system, this reference provides a convenient way for systems staff to identify the precise point at which the problem occurred.

The title area can be used to confirm the position in the system which the user has reached. In a menu hierarchy, the status area may indicate the path he has taken through the system by displaying the options chosen on previous menus, as in Fig. 7.8. It can also be used to provide confirmation that the system is still operating.

Optional *upper* and *lower message areas* provide consistent locations for messages which give instruction to the user or indicate exceptional conditions.

```
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. 7.9 Status and message areas in a spreadsheet structure.

Instructions which pertain to how the screen should be processed appear in the upper area; those which pertain to how it is disposed appear in the lower area. You need to know 'how to do it' before you start and 'what to do with it afterwards' when you have finished! Either area can be used for help or error messages. It is more common to display help messages in the upper portion of the screen and error messages in the lower. Messages requiring action by the user (e.g. to confirm the values input in form filling) would normally appear in the lower area. The use of an upper message area is well illustrated by hybrid dialogue structures. This area is used for the display both of command menus and of error messages (see Fig. 7.9)

The *screen body* contains the main information which the screen is seeking to impart. In a menu structure, it contains the list of options; the menu header might be considered to appear in either the upper message area or the screen body. In our form filling example, it is the area within which the form is displayed and the input fields echoed. In a hybrid dialogue, such as the spreadsheet illustrated in Fig. 7.9, it contains the cells of the spreadsheet.

One or two lines are reserved at the bottom of the screen to display *standard facilities* which are available on all screens. The message areas contain instructions specific to that particular screen body. A possible use of this area, to indicate the meaning of function key input, is illustrated in Fig. 7.10. It is called an *escape hatch* because it often contains a facility which enables the user to 'escape' from the normal processing flow.

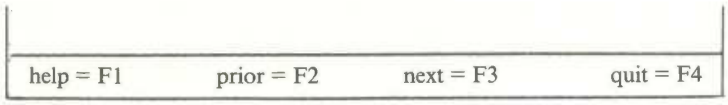


Fig. 7.10 The Escape Hatch.

This particular template splits the screen horizontally into a number of fixed 'windows'; it is a simple example of the techniques discussed in Chapter 10 for handling dynamic windows. The fact that the layout of every screen consistently conforms to *some* recognised template is more important than which particular template is chosen.

7.5.3 Positioning Error Messages

With a Question and Answer structure, the error message is normally displayed alongside or underneath the answer field. For example,

From: **dvx** Port code not recognised

An error message is usually unnecessary in a menu structure since the most likely cause of error is miskeying or a slip with a pointing device. If an error message is to be displayed, it usually appears below the list of options and in the lower message area.

In both the command language and the forms structure, a number of answers are input and so there are potentially a number of error messages. Most command language structures stop after detecting the first error and hence will display a single error message. In teletype mode, this message is displayed below the input line and the dialogue advances to the next line with a request for reinput. The display of error messages is slightly more problematical in a form structure. The user usually completes the form and returns to edit any fields in error. Therefore, the layout must accommodate the display of a variable number of error messages.

With a simple form, such as that in Fig. 7.11, it may be possible to display an error message alongside the field in error. This is the normal way to display a confirmation message, such as the description corresponding to a code.

This approach is seldom possible with a highly formatted form. In our ferry booking example, an attempt to leave space for a possible error message alongside each input field is likely to destroy the balance of the screen. It is improbable that sufficient space can be reserved in a single area, such as the

Invoice Number:	**[V12345]**	
Invoice Date	**[12/11/89]**	**Postdated Invoices not accepted**
Customer Code:	**[C23]**	Clough Brothers Ltd.
Invoice Amount:	**[100.00]**	
VAT Amount	: **[15.00]**	
Invoice Total	: **[151.00]**	**Invoice Total does not tally**

Fig. 7.11 A simple form with error messages shown alongside fields.

lower message area, to display all the possible error messages simultaneously. As we saw in Chapter 5, the system usually flags the input fields in error, positions the cursor at the first field in error and displays the corresponding message. As each input field is re-entered, the cursor moves to the next one in error and displays its error message.

7.6 Highlighting

Highlighting is the use of 'strong' attributes to make a particular area stand out from the rest of the screen and thereby attract the user's attention to it. Clearly the user's attention can only be attracted to a limited number of areas and if highlighting is overused, it becomes confusing rather than helpful. This is fine if the designer's objective (as in a video game) is to prevent the input process being too easy but it is unlikely to be desirable in most applications.

Initially the designer should specify a layout without any highlights and then go through each field asking himself what positive advantage would arise from the addition of a highlight. If the system only displays the information which is relevant to the user, there should be limited need for emphasis. Many existing systems demonstrate the temptation of highlighting; it is very easy to add just that bit more...

We have defined the attributes of a field in terms of a *foreground* colour, a *background* colour, a *bold* contrast level and a *blink* setting. These various features have different attention-getting powers; some are harder to ignore than others. To avoid diluting their impact, use only the minimum highlighting necessary to attract the user's attention. A person's attention can be attracted with a delicate nudge rather than a punch in the ribs, provided that they have not been subjected to constant battering! A combination of highlighting features is needed only when an exceptional emphasis is required.

A blinking foreground is the strongest visual attention-getter and hence

potentially the most distracting. It is best restricted to a single character position alongside the field to be highlighted. Blinking the text of a message is a good way of making it difficult to read!

Colour is the next strongest attention getter. Different colours in the spectrum have different *warmths* and *perceived brightnesses*. Areas shaded with backgrounds in the warmer colours at the red end of the spectrum appear larger than those shaded in colours at the blue end of the spectrum. Areas with backgrounds in white and colours towards the middle of the spectrum appear brighter and are easier to view under a wide range of ambient lighting. The best separation of two areas occurs when one is shaded in black or a colour from near one end of this spectrum and one is shaded in white or a colour from near the middle. The same consideration applies to distinguishing foreground content from the background of a field.

Users like the use of colour. Humans can distinguish many thousands of different colours but can cope with only a limited number at one time. There is also a danger in applying guidelines from colour printing to a screen. Much of the impact of colour printing arises from its use of subtle hues; the brash palette of colours provided by most colour screens does not permit such subtlety and there can be wide variations in the same colour produced on different screens. Certain juxtapositions of these colours, like a blue foreground on a red background, are positively unpleasant to the eye. The only real way of assessing how the colours will appear is to view them on the screen.

The implications of colour blindness can be overstated; it is typically an inability to discriminate between two very specific shades of these colours, accentuated by particular ambient lighting conditions. More important is the fact that all humans bring expectations of the meaning of different colours — red means stop, danger, etc.; colour coding within the system should be consistent with these expectations. A system which uses red for status messages that confirm everything is satisfactory and green for error messages is likely to be confusing.

The consequences of the above for the use of colour on a screen are summarised in Fig. 7.12.

On many monochrome screens, the effect of different colour attributes is to produce different shades of the screen's base colour; thus, on a green screen, different background colours might result in different intensities of green. The eye is less able to distinguish different contrast levels than different colours and the designer should beware of unwanted and nauseous highlights occurring when a system is ported from a colour to a monochrome device.

- Use the minimum of number of colours and not more than three or four on any screen.
- Use background colours in large blocks.
- Use bright colours for emphasis and weaker colours for background areas.
- To distinguish two areas of background, or to distinguish foreground and background, contrast black or a shade from one end of the spectrum with white or a second shade from near the middle.
- Use colour coding consistent with the user's expectation.
- Try it out on the actual screen.

Fig. 7.12 Guidelines for using colour.

Good results can be achieved with two levels, using the higher intensity background to 'box' the area to be emphasised. Inverse video is an example of this effect which is commonly used to highlight messages indicating exceptional conditions such as an input error; the area with the lighter background draws the user's eye. A number of upmarket workstations with black and white displays present a completely inversed image — printing is black on white! However, there is a danger of a 'shimmering' effect from large areas of inverse video which is very tiring on the eyes; it should be possible for a user to select normal or inverse presentation.

The use of different foreground intensities is the least intrusive attention getter. It is particularly effective for distinguishing data fields from output messages such as prompts and captions; the field to be emphasised has the bold attribute set on. A designer is usually spared any temptation to use multiple intensities since most screens support only two levels — bold off (normal) and on (high).

Other highlighting features are possible on some displays. The content of a field may be underscored or displayed in a variety of type styles or sizes; the characters of any given type font are specified as a pattern of bits which correspond to on/off settings of the pixels in a character position. Neither technique has the same impact that it does in hard copy and a multiplicity of fonts hinders rather than helps discrimination and may increase the sense of clutter on the screen.

Finally there is sound. Anyone who has sat in a room with a large number of terminals 'beeping' for input like hungry chicks, or in an amusement arcade full of video games endlessly repeating snatches of electronic Wagner, will appreciate the crass intrusiveness of sound as an attention getter. Sound is effective where it is really important to attract the attention of a user who may

not be watching the screen and is therefore not susceptible to a visual highlight. Tests on aircraft cockpit warning systems have shown that its effect is rapidly diluted. Frustrated composers should ensure that they have included an 'off switch' for silent running in their design.

There appears to be little need for highlighting in our ferry booking example. Blinking will be used only as a single character block cursor to indicate the current cursor position. Different foreground intensities will be used to distinguish input fields from captions in the screen body and from the other template areas; a similar effect could be obtained by using two colours but there is no real need for colour. Error input fields will be indicated by using a different background intensity or inverse video; these will be associated with the error message in the lower message area by using the same highlight for the error message. No sound will be used.

7.7 Producing a Draft Design

Having decided the information to be displayed, its format, how it is to be grouped on the screen and what highlighting is required, it is time to start producing some pictures.

A number of clerical forms to assist the design of screen layouts have been produced; these have merits in terms of providing a documentary record but all have a major drawback: what the layout looks like on paper bears little ressemblance to how it will appear on the screen. The form is covered in a matrix of squares indicating the character positions and, in most cases, the aspect ratio of the form differs from that of the screen. The font, which will probably be handwritten, will be different; in fact, the forms seem to encourage many people to use upper case exclusively. Furthermore, there is no way to create the effect of highlights. The only way to get a true impression is to display the draft design on the screen.

A proposed layout for the screen is illustrated in Fig. 7.13. Two lines are reserved at the top of the screen for titling information. There is no upper message area since it is assumed that the existence of a source document renders completion instructions redundant. The body of the screen contains the three logical groups necessary to confirm a booking, and also the passenger details; these logical groups are differentiated by position and spacing.

Captions are aligned within groups and input fields are delimited by bracketing; note how several captions have been changed from the initial

proposals in Fig. 7.5. A lower message area of three lines is reserved for the display, in a different background contrast, of error messages and confirmation by the system of the booking. The bottom line displays the interpretation of function/cursor keys.

After the completion of input, the screen appears as in Fig. 7.14. Input fields are echoed with the bold attribute set to 'on'. The port names are alongside the input codes. Coded input and the registration number are displayed in upper case regardless of how they were input. The lower message area contains a confirmation by the system that the booking has been accepted; this appears after the user has pressed the F1 key to confirm the input (provided no errors have been encountered). If the booking could not be accepted because of insufficient space, a corresponding message would be displayed and the cursor returned to the first input field.

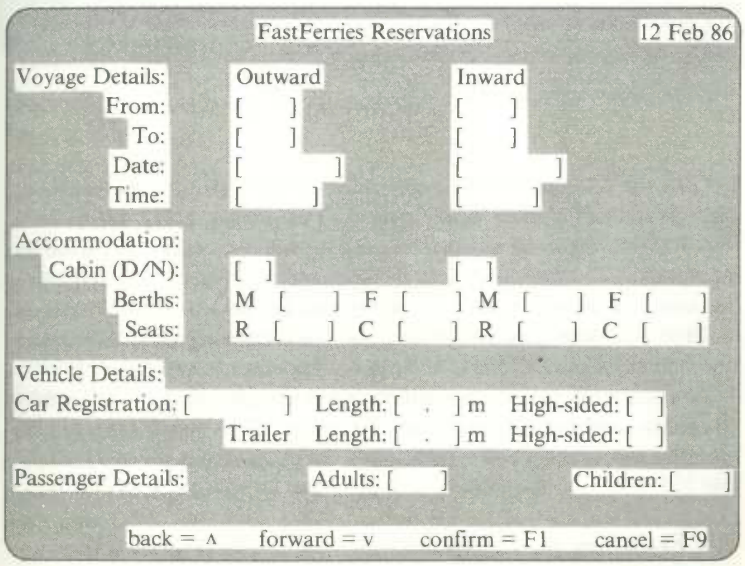


Fig. 7.13 A draft layout.

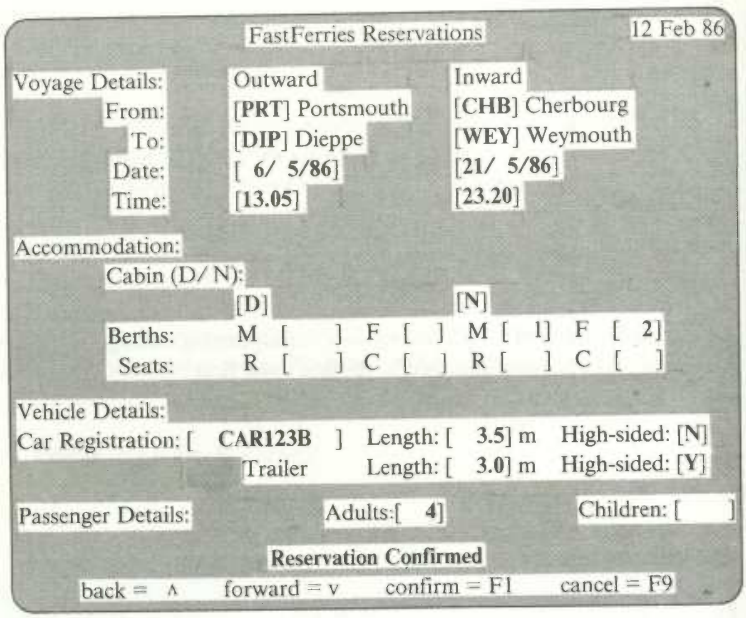


Fig. 7.14 The screen after completion of input.

Once the booking has been confirmed by the system, the screen body changes so that the other logical groups can be entered. This is illustrated in Fig. 7.15. Note that the group containing the voyage details is preserved for information, and that the options in the escape hatch have changed to allow the user to flip back to the prior screen and amend it if desired. Note the different conventions used for input of the type of camping reservation (which are mutually exclusive) and of the types of insurance (which are not).

As a final stage, the system will probably produce a summary which can be printed to provide a customer copy. This obviously no longer needs to have the same layout as the form. Its design is left as an exercise for the reader.

7.8 Evaluating the Design

These layouts are not presented as the optimum solution but merely as an

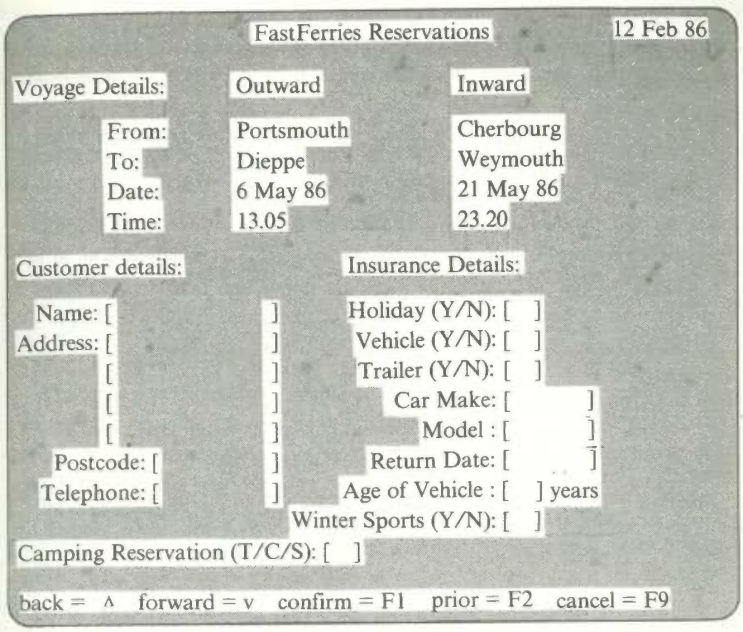


Fig. 7.15 The screen for the remaining logical groups.

illustration of the process; hopefully, they represent an acceptable starting point. The next stage is to evaluate the proposed design and repeat the process until an acceptable format is achieved. How can a screen layout be evaluated? Are there any objective measures which we can apply to test whether the screen is uncluttered, balanced and so forth?

The answer to this latter question is strictly neither yes nor no; a number of general mechanisms have been suggested but are not yet completely formulated. One problem is that the viewer of a screen is heavily influenced by the information content, which tends to cloud his judgement of the presentation. A general mechanism would divorce the content from the format; two suggestions for achieving this are boxing analysis and hot spot analysis.

Boxing analysis divides the screen up into physical groups; a group is an area of text characters with at least one blank space all around its perimeter. The smallest possible rectangle is drawn around each group, dividing the

screen into a set of boxes. Drawing axes about the centre of the screen gives an impression of the balance of the layout. The number and size of boxes gives an impression of the 'business' of the layout; a large number of small boxes suggests a 'fussy' layout.

Hot spot analysis attempts to identify the parts of the screen to which a viewer's eye would be drawn by the 'intensity' of the image at that point. The intensity at each point is computed as a moving average of the number of non-blank character positions around that point; increasing density is displayed by using characters which 'switch on' an increasing number of dots in the character matrix or increasing background intensities. One would expect a small number of hot spots symmetrically positioned about the central axes.

These mechanisms are described and illustrated in the reference quoted in the bibliography at the end of the chapter. However, even these techniques lack a quantitative measure: What is a 'large number' or a 'small box'? Is a point surrounded by the character '.' hotter than one surrounded by the character 'W'? These questions are yet to be answered.

Another argument against the general approach suggests that the attempt to divorce content from format is mistaken; a layout should be judged by fitness for its particular purpose rather than as an abstract piece of graphic design. By this token, the only way that the layout can be evaluated is by prospective users actually interacting with that screen. Although general techniques may be useful to a designer in eliminating some design flaws before the user evaluation takes place, there is no substitute for this type of evaluation.

Screen layout, like all aspects of the interface design, is an iterative process. A large number of layouts may need to be modified many times before an acceptable version of the system is produced. If the designer must produce individual code to generate an example of each layout, this will be an unacceptably lengthy process; an automated mechanism, a *screen design aid*, is required to support this.

7.9 Screen Design Aids

The output processes described in Chapter 3 go some way to reducing the effort of generating a screen layout. Rather than coding all the statements necessary to display output from scratch, the designer has merely to assign values to field data structures. A reader who has attempted any of the exercises will be assured of the savings this involves. However, it is still fairly time-consuming. To define a field the user must specify

- the content (not difficult);
- the slot; this means that the designer must already have checked this out roughly;
- the attributes; the effect of these must be a guess.

When this has been done for all fields, the code containing these assignments must be compiled, linked with library routines and executed. Any change in any aspect of any field definition means that the whole process must be repeated. Unfortunately, the designer is unlikely to have a real idea of what the screen looks like until the code is executed. It would be considerably easier if the designer could sit in front of a screen and 'paint' the proposed layout on it; he could then evaluate the screen's appearance as it develops. Figure 7.16 illustrates a screen design aid of this type.

The screen is divided into three parts. The upper message area contains information on the current cursor position and the current values of the attributes. The lower message area contains a menu bar which, among other things, allows the attributes to be changed. The screen body contains the screen that is being designed. The user enters text at any position by moving the cursor to that position and keying; this overwrites the character already there. Attributes can be changed by selecting Option 3 from the menu bar, and scrolling through the attributes in the upper message area, changing individual values to taste. To paint the attributes into an existing field, the designer positions the cursor to the start of the slot and presses a control key; another control key is used to indicate the end. The slot thus marked is redisplayed with the new attributes. The same effect can be produced with a mouse by pointing the mouse at the start of the field, pressing a button and dragging the mouse to the end of the field keeping the button pressed. New fields will take the existing values of the attributes.

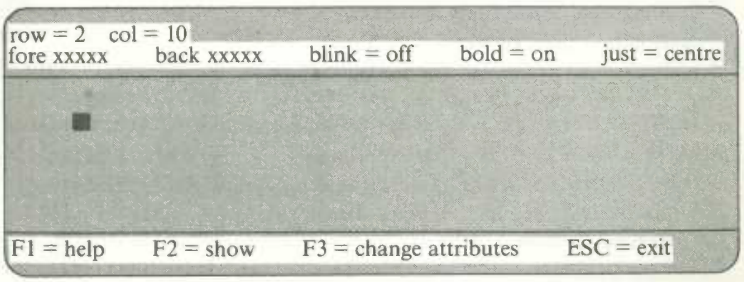


Fig. 7.16 A screen design aid.

Because some of the rows of the screen are taken up with information relating to the screen design aid, it would appear that no screen can be designed that uses these particular lines. One way round this problem is to allow the designer to scroll the screen body up and down, thus allowing the screen to be larger than the screen body. The full screen as it would appear in an application program can be displayed by selecting Option 2 of the menu bar. This is a toggle-switch; selecting it again redisplays the menu bars.

A screen design aid provides the designer with immediate feedback on the visual effect of the screen as he creates it. Changes can be effected simply by overpainting. The minimum requirement of a screen design aid is for it to provide facilities for creating, storing, retrieving and editing 'static' layouts. Further facilities are useful: a mechanism for naming the fields created by the screen design aid, and for linking these names to variables in the application program that is to use these screens; a mechanism for specifying the filtering and validation that is to take place at each field, and for automatically generating the appropriate code. Such facilities mean that the values of the various parameters in QandA_DIBs which define the dialogue can be specified by painting them on the screen. A number of proprietary packages exist which support some or all of these features and many installations have also developed their own. If such facilities do not already exist, their provision should be a high priority to any designer.

7.10 Summary

The screen design process involves:

deciding what information is to appear;
deciding how and where each field will appear;
deciding what highlighting is required;
developing a draft layout;
evaluating the effectiveness of the layout.

To ensure consistency, the overall layout of all screens should conform to a standard screen template. This template should identify the position of the information whose communication is the primary purpose of the display and of supplementary information (titling, status and instructions) which support the user's interpretation and manipulation of the primary information.

The screen should contain all the information needed by the user at that point and only that information. The information should be organised so that

its physical positioning reflects its logical grouping and is presented in an immediately usable format. The designer must understand the task in order to be able to assess these requirements.

A cluttered screen will impede the user's reception of the information which it contains; there are a number of standard guidelines covering the format and spacing of messages which are applicable. Where related information must be split across several physical screens, care must be taken that each screen still contains all the information necessary to process that particular screen.

Highlights attract a user's attention to particular areas of the screen and help to classify the information displayed in different areas. There are guidelines which can prevent the misuse or overuse of highlighting; intended highlights must still be tested on the screen on which they will be used because of wide variations in the way they are implemented.

Although some objective measures for evaluating screen designs have been suggested, these cannot replace trials with the intended users.

In order to support these trials and rapidly reflect their results in the design, a screen design aid is an invaluable tool.

Discussion Exercises

D1. Design the screen layouts which you feel are necessary to support the processing by a mail sale clerk of a telephone order from an existing agent. The layout(s) should contain explicit illustrations for the handling both of items which are in stock and of items which are unavailable. Illustrate how erroneous input will be handled and give examples of error messages and any help facilities which you feel are necessary.

D2. Several companies have expressed interest in the use of Prestel for direct sales to the public. A single product, 'Bert Blogg's Biggest Hits', available on record or cassette at the giveaway price of £3.99, will be advertised on TV. Members of the public, who subscribe to the Prestel service and have a full alphanumeric keyboard, can obtain any number of copies of the product by completing a Prestel response frame. Prestel allows full screen addressing, all the common highlighting features and simple graphics. It is designed to utilise a 20 line by 40 column colour TV monitor. If you have never encountered Prestel, an idea of the *display* facilities can be obtained by watching the Teletext services such as Ceefax or Oracle. For a Prestel response frame, input fields can be defined with simple validation checks.

Design the layout of a response frame which will enable users to order the product by specifying their name, address and credit card number (Visa, MasterCard and American Express are the only cards accepted).

D3. A software package is required to assist junior-school teachers in the development of vocabulary in slow learners; these are typically 9–10 year olds with a reading age of 5–6. The teacher will define a set of simple graphic representations (for example, a boat, a window or a flower) and will associate with each drawing a list of 5 words of 8 or less letters. A child sitting at a microcomputer will see the picture displayed and must choose the appropriate word. Pupils progress independently through a series of ten pictures.

The software will run on a typical home computer with a standard keyboard and 22 × 40 colour monitor. The pictures and wordsets will be stored on a disk. Design the screen layout which a pupil will see displayed and indicate how a word can be selected. Discuss what should happen if an incorrect choice is made.

Programming Exercises

P1. Determine the effect of different video attributes on the device you are using by producing a colour matrix which displays the possible combinations of background and foreground colour. The FieldType definition assumes 8 colours, so the matrix should be an 8 × 8 array of fields; the field in position (i,j) will have background colour[i] and foreground colour[j]. This exercise produces quite interesting effects even with a monochrome screen.

P2. Program the mail sale system for a single clerk. You need random access via a key to both the agents file and the products file. If your language supports indexed sequential files, then implement the files as:

Agents File	: organisation — indexed access method — dynamic
Products File	: organisation — indexed access method — dynamic

If your language only supports relative files, then use agency numbers of A1, A2, A3,... and have the file self-indexed (i.e. record A1 is stored at record position 1, record A2 at record position 2, and so on). Similarly for the products file, with product codes of P1, P2,...

Design your program so that the order is built up in memory, with the

individual lines of the order being stored in the elements of an array. Implement 'Send Order to Picking/Invoicing' simply as a procedure that prints the order in an appropriate format.

Further Reading

Bass L. J. (1985) 'A Generalized User Interface for Applications Programs', *Comm.ACM*, **28**, 6.

Galitz W.O. (1981) *Handbook of Screen Format Design*, QED.

Mason R. E. A. and Carey T. T. (1983) 'Prototyping Interactive Systems', *Comm.ACM*, **26**, 5.

Mehlmann M. (1981) *When People Use Computers*, Prentice Hall.

Pakin S. E. and Wray P. (1982) 'Designing Screens for People to Use Easily', *Data Man.*, July.

Smith S. and Mosier J. L. (1984) *Design Guidelines for User Interface Software*, MITRE Corporation.

Streveler D. J. and Wasserman A. I. (1984) 'Quantitative Measures of the Spatial Properties of Screen Designs', *INTERACT '84*.

Chapter 9

Simple adaptation

9.1 Introduction

We have seen that it is important for the dialogue to match the expectations and psychological limitations of the user; these depend both on the nature of the task being undertaken and on the nature and level of experience of the user. The use of hybrid dialogue structures, as discussed in Chapter 5, lessens the problem of a system encompassing a variety of different tasks. However, it is seldom sufficient to cater for the fact that any of these tasks can be undertaken by a variety of users whose levels of experience span the spectrum from beginner to expert. It is extremely desirable, therefore, that there be sufficient flexibility in the dialogue that it is capable of being adapted, or of adapting itself, to accommodate a potentially wide spread of user experience. Adaptability in dialogues can be categorised into three types: fixed, full and cosmetic.

With *fixed adaptation* the user explicitly chooses the level of dialogue support. The need to provide some type of adaptation was recognised early on; it is self-evident that a beginner needs more support than an expert, and early work focussed on fulfilling this particular need. It was typified by a *rule of two* in which a system provided two forms of dialogue:

a *verbose* form providing explicit support for the beginner;
a *brief* form aimed at the expert, offering little or no support.

A menu structure can be considered a verbose form, and a Q&A structure a brief form. The 'Rule of Two' has been extended to a 'rule of N', and a number of popular packages allow the user *explicitly* to choose one of N levels of dialogue. For example, one well known word processor provides four levels of help which can be set by the user. There are several limitations in this approach:

- It does not recognise a continuous spectrum of user skill; users do not change from beginner to expert in discrete jumps.
- A user may be an expert in one area of the system, and a complete novice in other areas.

- How does the system decide whether the user is novice or expert? One common approach is for the system to ask the user to choose the desired level at the outset. Surely any way of phrasing such a question is likely to elicit an answer which tells you more about the user's self-confidence than his competance!

With *full adaptation* the dialogue attempts to maintain a model of the user which changes with his usage of the system, and which controls the style of dialogue, adapting automatically as a result of these changes. But what predictors should the dialogue use when deciding about changes to the model? The time that it takes a user to respond? The number of errors he makes? The number of times he requests help? The nature of the errors and the type of help required? Recognising the characteristics of the user is one of the major problem areas in implementing adaptation in the dialogue; it assumes that the dialogue has a knowledge both of potential users and of the tasks. Much recent research has focussed on how the interface can develop and maintain models of its users as a basis for adaptation, utilising techniques of Artificial Intelligence.

Cosmetic adaptation seeks to provide flexibility of dialogue styles without either attempting to react to the user's behaviour or explicitly requiring him to select a particular style. It achieves this by providing shortcuts (called *accelerators*) such as

abbreviation and partial matching
synonyms
type ahead and answer ahead
default answers and macros
multi-level help

which an experienced user can utilise if he wishes. They are cosmetic in the sense that, like cosmetics, they represent essentially superficial enhancements to the basic structure but are, nevertheless, valuable in reducing tedium and in permitting limited personalisation of the interface. In this chapter we consider how such features can be incorporated into the dialogue.

9.2 Flexibility in Matching

As we have seen, selection inputs involve matching the user's input message against a list of possible targets represented as

```
type   TargetListType    = array[1..MaxTargets] of FieldType;
var    NumberOfTargets : byte;
       TargetList      : TargetListType;
```

The matching itself can be undertaken in a number of ways; the type of matching to be used at any point is specified by a parameter in the relevant QandA_DIB.

9.2.1 Normal Matching

The MatchString algorithm reproduced in Fig. 9.1 requires an *exact* match between the subject and one of the targets. It is the designer's responsibility to ensure that the targets are all unique. If they are not, then MatchString will select the first match.

9.2.2 Abbreviated Matching

Abbreviation has the obvious merit of reducing the volume of input. This might appear to favour inexperienced users who are likely to be less competent with the keyboard. However, abbreviation may impair the learning afforded to a novice through using the full input. Abbreviations tend to develop with usage — in the 1950s, that domestic novelty now known as the TV was still called a television. Some systems automatically *complete* the input i.e. they echo the complete target, rather than the abbreviation which the user entered.

```
function MatchString(subject:string;
                     TargetList:TargetListType;
                     NumberOfTargets:integer):byte;
var k,match : byte;
begin
match:=0;
k:=1;
while (k<=NumberOfTargets) and (match=0) do
   if TargetList[k].content=subject then match:=k
                                    else k:=k+1;
MatchString:=match;
end; {MatchString}
```

Fig. 9.1. Normal matching.

If natural item descriptions can be chosen so that a single character uniquely identifies each item, this is undoubtedly desirable. If, however, opaque or eccentric names or abbreviation rules have to be invented to permit single character abbreviations, any gain from the abbreviation is likely to be offset by a loss of naturalness.

Where abbreviations are permitted, there should be a consistent rule for creating them which pertains throughout the system. For example:

> a valid abbreviation is any number of characters greater than or equal to the minimum necessary to identify uniquely the item required.

Figure 9.2 shows the abbreviations by which some options may be identified. In abbreviated matching the number of characters in the input string governs how many characters take part in the comparison. Because of this, it may match more than one target. For example, the abbreviation 'p' would match both 'property' and 'personal'. So there are three possible outcomes of this matching:

no match	— does not match any of the targets
unique match	— matches one and only one target
ambiguous match	— matches more than one target

We will use a PASCAL 'set'

```
MatchSet : SetOfByte;
```

to record the ordinal numbers of the items of TargetList which matched the input string. If MatchSet is empty we have the 'no match' case; if MatchSet contains a single value, we have the 'unique match' case; and if MatchSet contains more than one value, we have the 'ambiguous match' case.

The AbbreviatedMatch procedure shown in Fig. 9.3 utilises the QandA_DIB, matching the content of the 'answer' field against the TargetList.

item		valid abbreviations					
motor		mo	mot	moto			
contents	c	co	con	cont	conte	conten	content
property		pr	pro	prop	prope	proper	propert
personal		pe	per	pers	perso	person	persona
life	l	li	lif				
miscellaneous		mi	mis	misc	misce	miscel	miscell–
end	e	en					

Figure 9.2

```
procedure AbbreviatedMatch(QandA_DIB:QandA_DIBtype;
                           var MatchSet:SetOfByte);
var k:byte;
begin
with QandA_DIB do
   begin
   MatchSet:=[];
   if length(answer.content)<>0 then
      for k:=1 to NumberOfTargets do
         if length(answer.content)<=length(TargetList[k].content) then
            if answer.content=copy(TargetList[k].content,1,
                                   length(answer.content)) then
               MatchSet:=MatchSet+[k];
   end;
end; {AbbreviatedMatch}
```

Fig. 9.3.

```
    if count(MatchSet)=0 then
       begin
       ChangeFieldContent(ErrorMessage,
          'Please re-enter - valid replies are ......');
       DisplayField(ErrorMessage);
       end
    else
    if count(MatchSet)>1 then
       begin
       ChangeFieldContent(ErrorMessage,
          'Choice is ambiguous - which of ......');
       DisplayField(ErrorMessage);
       end;

where
    function count(MatchSet:SetOfByte):byte;
    {returns the number of entries in MatchSet}
    var counter,k : byte;
    begin
    counter:=0;
    for k:=1 to MaxTargets do
       if (k in MatchSet) then counter:=counter+1;
    count:=counter;
    end; {count}
```

Fig. 9.4.

The input routine will keep requesting input until an option is 'matched', i.e. until a unique match is obtained. However, the error handling routine may well wish to distinguish between invalid and ambiguous inputs. In particular we may wish to display different messages to reflect these different types of error, as illustrated by the code fragment in Fig. 9.4.

Another form of abbreviation which is adopted for file names by many operating systems is the *wild card* technique. For example, both CP/M and MS-DOS interpret the string:

B:XX*.??Y

as identifying all files on drive B whose names start with the characters XX followed by any number of characters and with an extension which consists of any two characters followed by a Y. The matching process in such cases is more complicated than the example given and is more akin to parsing a command string.

9.2.3 Partial Matching

Abbreviation provides a mechanism for accelerating input and reducing potential typing errors. But it can hardly be considered a mechanism for error tolerance as discussed in Chapter 6. An error tolerant approach would cater in the matching process for simple typing or spelling errors. If MatchSet is empty at the end of the matching, the dialogue may simply tell the user that his response is incorrect, and ask him to re-enter it. Alternatively, it could try a 'partial' match on his current response.

On the assumption that errors are more likely to be made in the later input characters, you might progressively reduce the strictness of the match. One simple method is to delete the rightmost character of the 'subject', and then compare the new 'subject' with each of the match strings. This is repeated until either match(es) are found or 'subject' is reduced to a single character. This method may also produce an ambiguous match. A procedure for partial matching is shown in Fig. 9.5.

This partial matching algorithm has the advantage that it will still match correctly when the user types in more than enough uniquely to identify an item, and makes a mistake in the non-significant characters, e.g. by typing 'prpo' rather than 'prop'. Both the normal matching algorithm and the abbreviated matching algorithm will reject such a case as invalid.

```
procedure PartialMatch(QandA_DIB:QandA_DIBtype;
                       var MatchSet:SetOfByte);
var k:byte;
begin
with QandA_DIB do
   begin
   MatchSet:=[];
   while (length(answer.content)<>0) and (count(MatchSet)=0) do
      begin
      AbbreviatedMatch(QandA_DIB,MatchSet);
      delete(answer.content,length(answer.content),1);
      end;
   end;
end; {PartialMatch}
```

Fig. 9.5.

A number of more general approaches for implementing partial matching exist, all of which rely on establishing a metric so that the 'distance' of the input from each of the possible responses can be measured. The response(s) selected are those which are 'closest' to the input. Partial matching is particularly appropriate where the input consists of names (notorious for their eccentric spellings) or narrative input (for example, in searching a database by keyword) is involved. The Soundex system is one example of matching names based on their sound.

The name input is converted as follows:

remove all non-alphabetic characters (such as '–')
retain the first letter of the name
drop all vowels A E I O U and the letters W H and Y
assign the following numbers to the remaining letters:

1 = B F P V
2 = C G J K Q S X Z
3 = D T
4 = L
5 = M N
6 = R

coalesce adjacent identical numbers to a single number
form the code from the original first letter and the first three of the numbers; if less than 3 numbers remain, pad out with zeros

Thus FARBES, FFORBES, and FORBOUYS which all convert to F612, would all match an input of FORBES but so would FAIRPIECE. This illustrates both an advantage of such a mechanism and a drawback; it tends to result in ambiguous matches which require the system to ask for confirmation from the user. The Soundex system is clearly based on the 'phonetic' closeness of different characters; a similar 'metric' might be applied to the relative closeness of keys on a keyboard.

9.3 Synonyms

A dialogue which supports *synonyms* allows an object to be 'named' by a number of different identifiers, in a variety of formats, or by a variety of mechanisms. For example, MS-DOS allows both DEL and ERASE as

equivalent command names for deleting files and also allows DEL to be 'spelt' del. We have seen in Chapter 5 that it is easy to implement menus in which the user may select an option either by scrolling or by keying its identifier.

The merit of synonyms is that a user can use the identifier which seems most natural to him and hence the one which he is most likely to remember. The drawback is that this multiplicity can both confuse the user and complicate the processing required for matching; in particular it can slow down the response of the system if a large number of alternate names must be scanned. Another point to consider is whether there is much rationale for alternate identifiers unless the user can choose his own. This immediately leads to the problem of what happens if different users choose the same identifier for different objects, e.g. 'list' both to display on the screen and for printer output.

It is easy to build a limited and standard set of synonymous identifiers into the dialogue. One method is to associate a translation table with the TargetList of valid responses, as in Fig. 9.6. When the matching routine returns the ordinal of the choice, this is used as an index into the table; consequently both 'delete' and 'erase' would return option 2.

Another possibility is to use a cover routine for each synonym which calls the basic option. For example

```
procedure erase(files:filename);
begin
delete(files);
end; {erase}
```

	TargetList	Translation
1	copy	1
2	delete	2
3	erase	2
4	.....	.

```
CurrentTarget:=MatchString(subject,TargetList,NumberOfTargets);
if CurrentTarget<>0 then
        TranslatedTarget:=Translation(CurrentTarget);
```

Fig. 9.6.

One form of synonym which should always be supported (unless there is a specific reason not to do so) is to allow input in any combination of upper and lower case so that, for example, both 'property' and 'pRoPErTy' would match 'Property'. Novice users, and some not so novice, encounter problems with *case sensitive* input, by inadvertently leaving the Caps Lock key on. Even though UNIX is very generous with synonyms, it is case sensitive; for example, 'ls' causes a directory listing and 'LS' a search for a user defined object with that name. Of course, if you have already reserved most of the good names for the operating system, you may feel that you should restrict their influence! Converting an uppercase character to its lowercase equivalent, or vice versa, is trivial since

ord(UpperCaseChar) = ord(LowerCaseChar)+N

where N is an integer which depends on the particular character set.

A single space (or a combination of spaces) is often used as a delimiter to separate the various elements of a command language string. Input should be insensitive to the actual number of spaces entered, and so multiple spaces should be reduced to a single space. (There are exercises at the end of the chapter to convert case and to remove redundant spaces.)

9.4 Defaults

The essence of a *default* value is that the system will assume a particular response unless the user specifically types a different one. Thus, in cases where the user frequently enters the same answer to a particular request, the input can be reduced to the single keystroke necessary to confirm the default value.

Situations requiring a standard answer arise frequently. For example, when entering transaction data into an accounting system, a unique posting reference number and posting date are usually supplied. The reference number is typically the next integer in sequence and the date is typically the current date; it is unnecessary for the user to input these explicitly unless they are different from this assumption. Similar examples apply in command languages. The command DIR in CP/M and MS-DOS lists the directory of the current drive unless the user explicitly specifies a different drive.

The user must be aware of what default value the system is assuming. In Q&A and form filling structures, the default can be displayed with the prompt or field caption

Posting Reference [**1047**]?
Posting Date [**26/10/85**]?

or in the position which the response would occupy

Posting date	**26**	**10**	**85**

Since some users find overtyping existing values confusing, the ReadField keyboard process of Chapter 5 displays the default in the answer field, but clears the field as soon as the user enters the first character of the reply.

In a menu structure, the default is usually indicated by its position (the first item on a menu organised by frequency of use) or by a video highlight. In a scroll menu, moving round the options could be considered as temporarily changing the default value.

In a command language, there is no way to display a default value and so the user must remember it. However, since a command language is aimed at experienced users who have to remember the command syntax anyway....

The commonest way for a user to accept the default value is to enter a null input, i.e. just press the 'Carriage Return' or 'Enter' key. If command language with positional parameters is being used, the user simply omits the entry, which results in two adjacent separators

MODE COM1:9600,,8,1

This would be interpreted as a null reply for the fourth token, leading to the default value (even parity in this example) being used. In a command language with keyword parameters, a default value is selected simply by omitting the relevant parameter keyword and value.

Since users often get the habit of pressing the return key almost automatically, the default value in cases where the outcome is irreversible is customarily the one that preserves the status quo. Thus NO would be the default response to

A>**era** *.*
Sure [Y/N]?

Defaults assume that one answer will be the most common; it is foolish to implement a default value which is constantly being overridden. A major problem is that, like everything else in the dialogue, the most common answer is likely to change with the user and over time. Thus there must be some mechanism for changing the default values within the system; if these values

are coded into a program source such changes will be very tedious! The reader may care to consider how this might be overcome; we will return to it in a later section.

The dialogue processes of Chapters 4 and 5 use the following mechanism for defaults. The default answer for a particular QandA_DIB is stored in the answer field of that DIB. When the Q&A procedure is invoked to accept input via this DIB, the current content of the answer field is displayed. If the AcceptField key is pressed immediately, this default content is accepted; pressing any other valid key causes the field to be cleared, and input which falls within the filter is accepted and displayed in the field.

The user may not just provide the same answers to isolated questions; his usage of the system may be such that he commonly traverses the same paths, i.e. he regularly inputs the same sequence of answers. For example, he may often select a report using the same selection criteria. This could be achieved by repeatedly pressing the return key to accept each default.

```
Report No [1]   : <CR>
Products [ALL]: <CR>
Markets [EUR] : <CR>
Sources [UK]    : <CR>
.....
```

but it would be tedious. What is needed is a mechanism for 'cataloguing' a series of standard responses such as that provided by operating systems, e.g. BAT files of MS-DOS. *Macros* provide such a mechanism. The user stores his responses for selecting a commonly used report under some name, e.g. MYREPORT. The user responds to the first prompt with the name of the macro (and some indication that it is a macro) and the stored answers are individually fed to each prompt and processed as though they came directly from the keyboard:

```
Report No [1] : *MYREPORT
```

How this may be accomplished should become apparent in the next section.

9.5 Type ahead and Answer ahead

9.5.1 Type ahead

We saw in Chapter 3 that, for most devices, if the user presses a key while the

CPU is executing a task process, the system will generate an *interrupt* which will:

- cause the CPU to suspend execution of the task process;
- start execution of the keyboard I/O process; this will take the character typed, store it in the keyboard buffer, and on completion of this;
- cause the CPU to resume execution of the suspended task process.

Users who follow regular paths through a system soon become familiar with the sequences of questions and possible responses. Suppose the user types m o t o r <carriage return> while a task process is executing. On completion of the task process the buffer will contain:

m	o	t	o	r	<CR>	

If the next request for input requires the user to select from a menu of 'motor, contents.....,end', the options will be displayed and the user asked to choose. However, the system will find the answer already in the buffer since the user typed it ahead of being asked (hence *type ahead*) and will continue immediately. There is little point in subjecting the user to a display of the menu or prompt if his answer is already in the buffer. Ideally this would only apply to complete answers; a partial answer would be processed in the usual way. Figure 9.7 illustrates the type ahead mechanism.

The advantage of type-ahead is that the user can form his own 'input closures' independent of the timing of prompts from the system. One

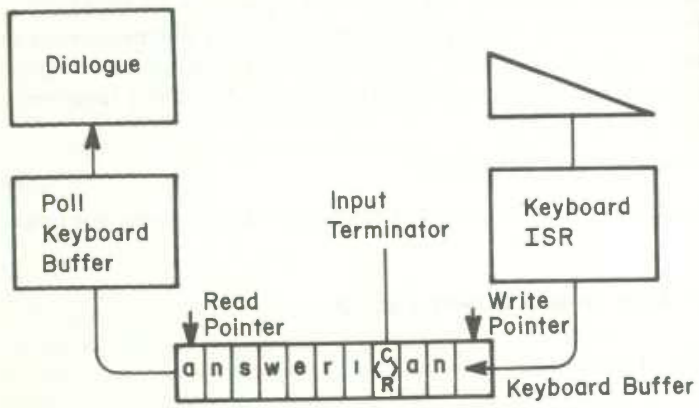
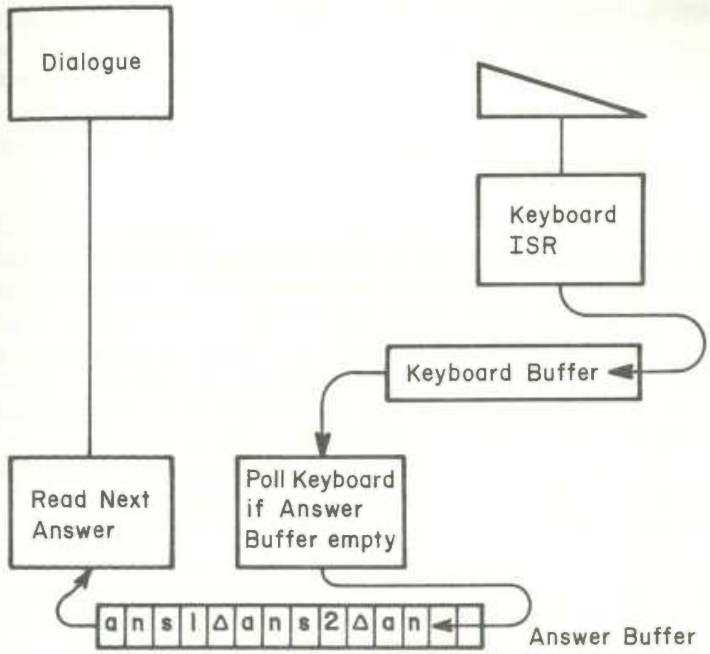


Fig. 9.7. Type ahead.




```
procedure GetReplyFromBuffer(reply,mode,buffer);
{mode = 0 if variable length replies are delimited by separator
 mode = m > 0 if replies of fixed length m are expected}
begin
if mode=0 then {variable length replies}
   {locate next answer}
   begin
   k:=pos(buffer,separator);
   if k=0 then {no separator, so take all}
      begin
      reply:=buffer; buffer:='';
      end
   else
      begin {separator}
      reply:=copy(buffer,1,k-1);
      buffer:=copy(buffer,k+1,length(buffer)-k);
      end
else {fixed length replies}
   begin
   reply:=copy(buffer,1,mode);
   buffer:=copy(buffer,mode+1,length(buffer)-mode);
   end;
end; {GetReplyFromBuffer}
```

Fig. 9.8. Answer ahead.

difficulty is that the type ahead buffer provided by the operating system may be relatively small, for example 15 characters, thus limiting how far ahead the user may type. When the buffer becomes full, excess characters are ignored.

9.5.2 Answer ahead

The user is not restricted to answering only the next question; a number of sucessive replies may be provided. This is called *answer ahead*. If the replies are not of a fixed length, each reply must be delimited by a *separator* such as Carriage Return or '/'. The separator must be chosen with care since it must not occur in the normal input stream. To ease typing, it should ideally be a single unshifted character (or at least a single keystroke). Since the keyboard buffer is normally of very limited capacity, a *dialogue answer buffer* is needed to implement answer ahead; the dialogue process will take each answer in turn from the buffer by locating the next separator, as illustrated in Fig. 9.8. Note that there is a difference between type ahead and answer ahead in that the dialogue answer buffer is filled only when the system requests input.

A further problem with answer ahead occurs if the user enters an invalid input in the middle of the buffer. It would be possible to provide a mechanism for editing the buffer after it has been input — many interactive systems provide a means to recall the last line input, edit and then resubmit it — but in many cases this is more cumbersome than merely clearing the buffer and restarting from that point. The buffer should also be cleared if the user requests help.

Additional routines are needed to support type ahead and answer ahead. In order to decide whether the prompt should be suppressed, the dialogue process must be able to test whether the user has typed/answered ahead:

function KeyboardEvent:boolean;
{returns 'true' if keyboard buffer is not empty, otherwise 'false'}

function DialogueAnswerBufferEmpty:boolean;
{returns 'true' if the dialogue answer buffer is empty, otherwise 'false'}

If a type/answer ahead is incorrect, the dialogue must be able to flush the remainder of the buffer

procedure ClearKbd;
{clears the keyboard buffer of any characters that may have been typed ahead of them being requested}

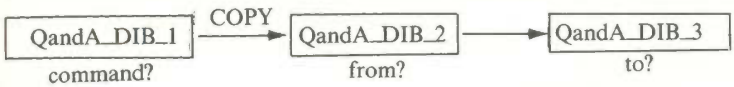
procedure ClearAnswerBuffer;
{clears the dialogue answer buffer}

9.6 Command Language as Answer ahead

Suppose we have a QandA_DIB which prompts for a command name:

command?

The user input is matched against TargetList which holds the permitted commands. When matched, a sequence of QandA_DIBs is processed to accept the parameters for that command. For example:



A typical dialogue might be:

command? **COPY**
from? **A:FILE1**
to? **C:FILE2**

Compare this with the following 'command language' string:

command? **COPY bb A:file1 bbbbb c:FILE2**

If the string is preprocessed to a standard format by converting all letters to the same case (e.g. upper case), and removing superfluous blanks, we obtain:

This can be considered as an answer ahead to the previous sequence of

QandA_DIBs with space as the delimiter. Because the answers to the 'from?' and 'to?' questions have been provided ahead of the questions being asked, these prompts have been suppressed. Hence, a command dialogue can be viewed as *question-and-answer with answer ahead.* The example chosen has used positional parameters but a similar argument will handle keyword parameters and the inclusion of 'switches' in the command string. Recognition of this fact can guide us when designing a command language.

With a command language, the user has the sensation of *being in control*: – the dialogue is user-driven. Because answer-ahead places greater demands on short-term memory, consistency and memorability are extremely important:

- The names assigned to commands should be meaningful, and easy to remember. Wherever possible, congruent pairs of names should be used (for example, GET and PUT, INSERT and DELETE, and so on).
- A standard delimiter (such as space or comma) should be adopted, and used consistently. Thus, the designer should avoid capricious differences in syntax such as

```
pip newfile=oldfile
ren newfile oldfile
```

- A standard escape convention should be used.

In the hands of experienced users, a well designed command language can lead to fast interaction; many computer professionals are very skilled at interacting with operating systems via command languages. However, it can take a long time to become proficient, and relearning can be a major exercise after an extensive interval away from the system.

If the command language is designed on the basis of Q&A with answer ahead, then it can cater for a range of user skills. For an experienced user, 'closure' will occur at the end of a complete command string; for an inexperienced or 'rusty' user, it may occur within a command string. If the user types only part of the answer, (perhaps because he doesn't know the full format of the command, or perhaps because he has forgotten it after some time away from the system) then some questions will be unanswered, and so the system can prompt him for the remaining answers. For example, if the user types:

command? **COPY**

and then presses Carriage Return, or a Help key system or pauses for a given

interval, the system can prompt with a form requesting the remainder of the information:

from? **A:FILE1** to? **C:FILE2**

By regarding it as a special case of Q&A, a command language can be made more flexible; in fact, we have introduced an *implicit* Rule of two. On the one hand, experienced users using answer ahead see the dialogue as a traditional command language; on the other hand, novices can be given the syntax of the commands, and specific help can be provided for each parameter. This mechanism can be extended further to an implicit rule of N. The possible values for command parameters could be scrolled through each answer box; a user may either type in a parameter value or scroll through the possible options using the supressed menu technique described in Chapter 4.

9.7 Multi-level Help

We saw in Chapter 6 that a good help message must be specific to the user's problem. This implies that a series of help messages are necessary at each point where input is requested from the user, rather than a single all-embracing message.

As a minimum, the system should distinguish between a user who has merely forgotten the precise 'phrasing' of the response (i.e. who requires a menu of the available options or formats) and a user who does not understand what input is being requested. This distinction can be achieved by implementing two different help requests. For example, an input of MENU displays the available options (the TargetList from the QandA_DIB), and an input of HELP provides a message which displays not only the options but a brief explanation of what each does.

However, like the rule of two, this is a rather crude distinction; a more subtle mechanism would allow N levels, although not all N would necessarily be appropriate at all points in the dialogue. Equally, the systems designer is likely to feel that there is some N above which more dramatic assistance is needed than can be supplied automatically.

One method of supplying multi-level help is based on displaying a first-level help message and then asking the user if he requires more. This process is repeated until either the user is satisfied or there is no more help that can be given. The sample dialogue at the start of Chapter 6 uses this approach which effectively introduces a Help sub-dialogue with a Q&A structure.

Option? : **HELP**
.....first level help .....
.....
More? : **Y**
.....second level help.....
.....

A more elegant mechanism relies on the system tracking the number of times the user has requested help at that point in the dialogue, for example by maintaining a CurrentHelpLevel, as shown in Fig. 9.9.

```
if reply=help then
   begin
   if CurrentHelpLevel>MaxHelpLevel then
      begin
      ChangeFieldContent(HelpMessage,'no more help available');
      DisplayField(HelpMessage);
      end
   else
      begin
      GiveHelp(StartAt[CurrentHelpLevel]);  {See Figure 6.5}
      CurrentHelpLevel:=CurrentHelpLevel+1;
      end;
   Buffer:='';   {can't answer-ahead!}
   end;
```

Fig. 9.9. Multi-level help.

The mechanism described in Section 6.4.2 can be easily extended to incorporate multi-level help. Instead of containing a single value N pointing at the first record of a single help message, the help message parameter in the QandA_DIB would contain an array of first-record pointers, one for each level of help, and its own current and maximum help levels.

```
CurrentHelpLevel : byte;
MaxHelpLevel     : byte;
StartAt          : array[1..GlobalMaxHelpLevel] of integer;
```

9.8 Multi-language Considerations

Amongst the earliest people to recognise the desirability of separating the interface from the task processing in a system were systems designers working in multinational companies in the 1960s. Systems which had been developed at great cost in North America or the UK were appropriate for operations elsewhere in the world but were difficult to transport because the prospective users, who were not fluent in English, found difficulty in understanding the

messages. Changing these messages was difficult as they were scattered as literals throughout the programs making up the system. Therefore a programmer was needed to go through the source programs, locating and changing these to produce a new source for the European location; of course it helped if this programmer knew French, German, Italian... or alternatively if there was a translator who knew COBOL.

Life would have been much easier if all these messages — prompts, help messages, error messages — had been held on a data file divorced from the syntax of the programming language involved. Then a translator could have been given a listing of the file to translate, and data preparation staff could have keyed in this translation to create another file. The literals could be referenced in the program via a number, for example:

```
Retrieve(MessageFile,MsgNo,Message);
DisplayField(Message);
```

There is litle point in displaying output in the user's native language if he still has to respond in English; the valid responses must also be held on file for ease of translation.

Ledger? : **sales** ====> Livre? : **clients**

We have already seen the desirability of holding help messages on file. Combining this with the multi-language capability requires a file which contains a record or group of records associated with each input request which details

the prompt message
the series of help messages
the TargetList of valid responses
the error message to be displayed if the response is invalid.

Since we have abstracted the parameters associated with any input request into a QandA_DIB data structure, this is equivalent to saying that the data values associated with each QandA_DIB must be held on file. Records are also needed for the other types of message, e.g. output data messages such as report titles, status messages, etc. The structure of the dialogue procedure can be adjusted to reflect this requirement, by including a routine to set up the DIB associated with any input step from the relevant file.

This mechanism enables the designer to allow for personalised dialogues. We have considered it from the point of view of providing a native language facility in systems which are to run at a variety of installations. However it

need not be restricted to providing the same dialogue in English or French or some other language. There is no reason why, where a clerk and a manager both use the same system for different functions, a different dialogue should not be provided merely by supplying a different data file which will run with the same dialogue process and the same background tasks.

A user can be restricted to a particular subset of the system's facilities by defining only that subset on the dialogue file. For example, a user who is allowed to read but not amend a personnel database would never see any reference to updating facilities, since the TargetList on his dialogue file would not contain the response 'update'. Different default values can be supported for different users and even the default paths described in Section 9.4 could be stored on this file.

On a terminal-based system, the required file can be identified automatically by some mechanism such as a log-in code, and on a stand-alone based system by providing the user with a diskette containing his personal dialogue. A little thought should indicate the undesirability of asking the user which dialogue he requires: if it is a choice of language, in which language do you ask for the choice?

It is also possible that the character used in answer ahead, and ControlDIB values such as the 'standard' response which users input to request the default or to request help, may vary for different users. These can also be held on the dialogue file but since they are the same at each input request, they do not need to be stored for every individual request and could be held in a header record at the start of the file.

The mechanism is not without drawbacks. We have already mentioned the problem of efficient storage. There will also be some degradation in response since a number of retrievals from file will be needed to set up the messages. In Chapter 11 we discuss how these reservations can be overcome, and consider how all the various dialogue structures and facilities can be accommodated in a generalised dialogue manager.

9.9 Summary

The dialogue must incorporate some element of adaptability to cater for varying levels of expertise in different users and in the same user in different parts of the system.

Fixed adaption is a crude mechanism since it requires the user to choose a particular level of support explicitly. Full (or automatic) adaption is

extremely difficult to implement since it requires the system to recognise a user's characteristics from his inputs. Cosmetic adaption provides accelerators which can, but need not, be utilised by an experienced user and provide a limited amount of personalisation.

Rather than requiring an exact match with an entry in the TargetList, the system may accept abbreviated or partial matches. Partial matches provide some error tolerance but it is often difficult to define a metric for deciding which target has been matched and such metrics often result in ambiguity.

Synonyms allow a user to identify system objects by a variety of identifiers, in a variety of formats or by a variety of mechanisms. They permit personalisation of the dialogue but may confuse rather than assist novice users.

Defaults reduce the input load by enabling a user to select a common response or set of responses (a macro) with a single input.

With type ahead and answer ahead, a user may supply a series of responses without waiting for the system to prompt for them individually. This allows a user to form his own closures rather than having closures imposed by the system.

Interpreting a command language structure as a Q&A structure with answer ahead, suggests both guidelines for the design of command languages and ways in which such a structure can adapt to cater for both novice and expert users.

A multi-level help facility is necessary to provide support which is specific to a user's problem. Such a mechanism is easily implemented by storing pointers to the relevant message levels in the QandA_DIB.

We saw earlier that developing a common abstraction (the QandA_DIB) facilitated the production of generalised processes which could implement any dialogue structure. Holding the values for these data structures on a dialogue file enables personalised dialogues to be produced for any group of users.

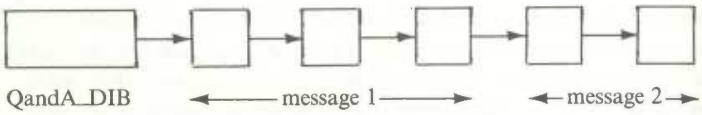
Discussion Exercises

D1. Two common keying errors are:

- the transposition of 2 letters e.g. TSET for TEST;
- the substitution of one letter by another e.g. TWST for TEST, often by hitting an adjacent key to the one intended.

Suggest a possible partial matching metric which could be used to handle these cases.

D2. An alternative way of implementing multi-level help is to superimpose multiple messages onto the structure described in Section 6.4.2 (a single pointer in the HelpMessage field of the QandA_DIB pointing to the first record of the first-level help message).



Messages can be assumed to occupy an integral number of records. However, some mechanism is needed to distinguish between the various messages. Initially the QandA_DIB pointer points to the first record of the first message. After one call for help, this pointer will be left pointing at the first record of the second message. And so on.

What are the advantages and disadvantages of this approach?

D3. Dialogue files provide a mechanism for supporting personalised dialogues for different users. Discuss the record format which is needed for such a file.

Programming Exercises

P1. Implement a routine to convert all ASCII lowercase alphabetics in an input string to uppercase.

P2. Implement a routine which will remove all redundant spaces from an input string, i.e. convert all multiple spaces within the string to single spaces and remove any leading or trailing spaces.

P3. Many operating systems support a macro facility whereby a series of command lines can be stored in a 'batch' file. When the batch file is invoked, the sequence of commands is executed as though they had been typed at the keyboard.

Suggest how such a facility can be used to support synonyms; for example, to provide a variant of the intrinsic file copy function by one with a different name and different parameter order.

P4. Amend the ReadField keyboard process to incorporate an answer-ahead .

mechanism which is independent of the operating system keyboard buffer. The dialogue process will maintain a global variable as a buffer. (How long should this buffer be?) Whenever a user is requested for input, he may type as many successive answers as he likes; this input string is read into the buffer. A prompt will be displayed and input requested only if the buffer is empty. Assume that the user terminates each input with a carriage return.

P5. If in Question P2 all the valid responses were single character replies, it would be silly to insist on a carriage return being typed either after a single reply or after a string of answer-ahead replies. How else could the system determine if the user had completed entering the answer-ahead string?

Further Reading

Benbasat A. *et al.* (1984) 'Command Abbreviation Behaviour in Human–Computer Interaction', *Comm.ACM*, **27**, 4.

Durham I. *et al.* (1983) 'Spelling Correction in User Interfaces', *Comm.ACM*, **26**, 10.

Good D. M. *et al.* (1984) 'Building a User Derived Interface', *Comm.ACM*, **27**, 10.

Mozeico H. (1982) 'A Human/Computer Interface to Accommodate User Learning Stages', *Comm.ACM*, **25**, 2.

Schneider M.L. *et al.* (1984) 'An Experimental Evaluation of Delimiters in a Command Language Syntax', *Int.J.Man–Machine Studies*, **20**, 6.

Wasserman T. (1973) 'The Design of Idiot-Proof Interactive Systems', *AFIPS*, **42**.