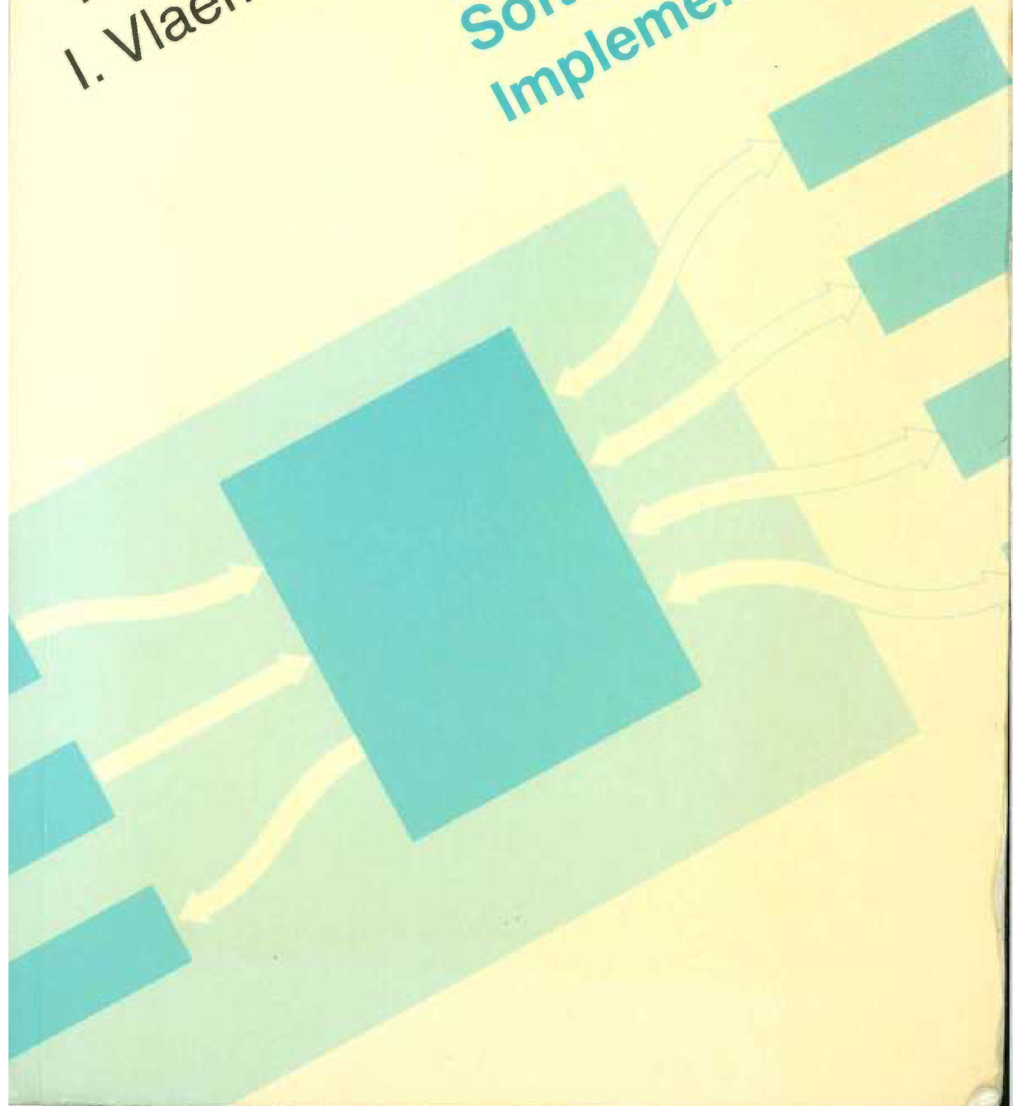
Blackwell
Scientific
Publications
Computer
Science
Texts
R.B. Coats
I. Vlaeminke
Man-Computer
Interfaces:
An Introduction to
Software Design and
Implementation

Man-Computer Interfaces Coats and Vlaeminke

Man-Computer Interfaces: An Introduction to Software Design and Implementation

This book provides an introduction to the basic concepts and facilities of text-based interfaces. The major underlying theme is that in any system, the interface software can be identified as a distinct element separate from the task processing. The book illustrates how this separation can be achieved and how interface functions can be represented by general abstractions independent of the devices on which the system is to be implemented. Most chapters incorporate sets of discussion and programming exercises. The book is suitable for an undergraduate course in Computer Science and for practising system designers.

THE AUTHORS

R.B. Coats, BSc, PhD, and I. Vlaeminke, BSc, MSc, are both Principal Lecturers in the School of Mathematics, Computing and Statistics at Leicester Polytechnic.

Software Reliability: Achievement and Assessment
B. Littlewood
0 632 01573 X (hb)

Occam Programming – A Practical Approach
J. Kerridge
0 632 01658 2 (hb)
0 632 01659 0 (pb)

Software Engineering: Principles and Methods
B. Ratcliff
0 632 01752 X (hb)
0 632 01459 8 (pb)

Modula-2: Constructive Program Development
P.A. Messer and I. Marshall
0 632 01609 4 (hb)
0 632 01508 X (pb)

An Introduction to Programming with SIMULA
R. Pooley
0 632 01611 6 (hb)
0 632 01422 9 (pb)

BLACKWELL SCIENTIFIC PUBLICATIONS LTD
Osney Mead, Oxford OX2 0EL
8 John Street, London WC1N 2ES
23 Ainslie Place, Edinburgh EH3 6AJ
52 Beacon Street, Boston, Massachusetts 02108, USA
667 Lytton Avenue, Palo Alto, California 94301, USA
107 Barry Street, Carlton, Victoria 3053, Australia

ISBN 0-632-01542-X

9 780632 015429

COMPUTER SCIENCE TEXTS

Man–Computer Interfaces

R. B. COATS
and
I. VLAEMINKE
Both Principal Lecturers
in Computing
Leicester Polytechnic

BLACKWELL SCIENTIFIC PUBLICATIONS
OXFORD LONDON EDINBURGH
BOSTON PALO ALTO MELBOURNE

© 1987 by
Blackwell Scientific Publications
Editorial offices:
Osney Mead, Oxford OX2 0EL
(*Orders*: Tel. 0865 240201)
8 John Street, London WC1N 2ES
23 Ainslie Place, Edinburgh EH3 6AJ
52 Beacon Street, Boston
Massachusetts 02108, USA
667 Lytton Avenue, Palo Alto
California 94301, USA
107 Barry Street, Carlton
Victoria 3053, Australia

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior permission of the copyright owner.

First published 1987

Set by V & M Graphics Ltd,
Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

DISTRIBUTORS

USA and Canada
Blackwell Scientific Publications Inc
PO Box 50009, Palo Alto
California 94303
(*Orders*: Tel. (415) 965-4081)

Australia
Blackwell Scientific Publications
(Australia) Pty Ltd
107 Barry Street
Carlton, Victoria 3053
(*Orders*: Tel. (03) 347 0300)

British Library
Cataloguing in Publication Data
Coats, R. B.
Man–computer interfaces: an introduction to software design and implementation.
1. Computer interfaces
I. Title II. Vlaeminke, I.
004.6 TK7887.5

ISBN 0–632–01542–X

Library of Congress
Coats, Robert B.
Man–computer interfaces: an introduction to software design and implementation/R. B. Coats & I. Vlaeminke.
Includes index.
ISBN 0–632–01542–X
1. Computer software—Development. 2. Electronic data processing—Psychological aspects. 3. System design.
I. Vlaeminke, I.
II. Title
QA76.76.D47C63 1987
005—dc19 87–16110

Chapter 3

Input/output processes

3.1 Introduction

In Chapter 2, the messages exchanged between a user and the system were classified in terms of the functions they performed within the dialogue. In this chapter, we concentrate on how the nature of the information exchanged impinges on software design. For this purpose we classify the basic input/output processes which support these message functions into the categories of Fig. 3.1.

- Output of a text message
  - at the current postition on the device
  - at a particular position on the device
  - with particular format attributes
- Input of a text message
  - using standard high level input routines
  - character-by-character
  - where 'special characters' are involved
- Input of a Point and Pick message
  - relative pointing to scroll round a list
  - absolute pointing anywhere on the device
- Output of a graphical message
- Input of a graphical message

Fig. 3.1. A classification of common input and output processes.

A *text message* is defined as a string of *characters*; these characters may be upper or lower case alphabetics, numerals, or even the basic graphics characters (such as playing card suit symbols) available as alternate character sets on many microcomputers. A *graphical message* is defined as a message which cannot be represented as a string of characters; typically it must be described at the 'bit' rather than at the 'character' level. *Point and pick* is a special case of an input message for selecting from a set of possible options; as we shall see, it has characteristics which deserve individual treatment.

The simple dialogue examples in Chapter 2 utilise the standard PASCAL input and output procedures for the input/output processes. In this chapter

SKYHAWKE Ex. 1016, page 6

we consider other features which are available to enhance these processes. These features have typically been considered in terms of the processing required to effect them on a particular device. However, we also saw in Chapter 2 that the dialogue process deals with logical entities and should be separated from the precise mechanics of a particular device. Therefore, whilst considering how the features can be effected, we will attempt to develop generalised 'abstractions' for the processes which the dialogue process can assume.

3.2 Output of a Text Message

3.2.1 Output of Text at the Current Position of the Device

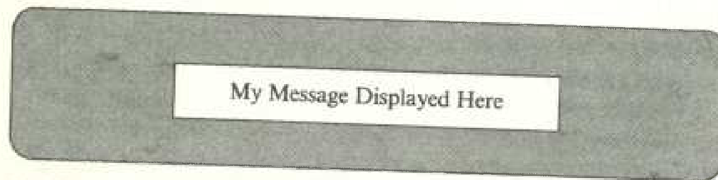

Fig. 3.2.

An output text message, like that illustrated in Fig. 3.2, can be specified by defining:

what is to be output, i.e. a string of characters identifying the content of the message;
where it is to be output, i.e. the position on the output device;
how it is to be output, i.e. a list of attributes, such as colour, which define how the content is to be formatted; we will assume that the attributes are constant for a given message.

We define an output message as an entity which is represented by a data structure of type:

```
OutputMessageType = record
                      content    : string;          {what}
                      slot       : SlotType;        {where}
                      attributes : AttributesType;  {how}
                    end;
```

Content is of type 'string' by virtue of our definition of a text message.

However, the definitions of SlotType and AttributesType require further consideration.

The most common input/output device in the early days of interactive computing was the *teletype*, which resembles a golfball typewriter. A teletype only provides the facility to output the message content at the current position of the device; the current position can be advanced to a new line by the output of suitable control characters. Most programming languages contain procedures for teletype operation.

The WRITE and WRITELN statements of PASCAL operate in this way; thus,

write ('write this without advancing to a new line')

displays the specified character string and leaves the output device positioned at the next character position and

writeln ('write this and skip to a new line')

displays the character string and positions the output device at the first character position on the next line.

The teletype has largely been superseded by the Visual Display Terminal (VDT) consisting of a monochrome or colour display and keyboard. Early VDTs operated merely as 'glass teletypes' with the same *line* scrolling mode. Nowadays, the vast majority provide *page* mode operation, in which the output can be considered a screen at a time rather than a line at a time; a message can be written at any position on the screen.

We will concentrate on output to a screen since it is the most common device; similar considerations apply for a printer. The basic approach is also appropriate for other output devices, such as a speech synthesiser, although the features will obviously differ.

3.2.2 Output of Text at a Particular Position on the Device

The text of an output message is displayed in a slot — a rectangular area on the device w character positions wide and h character positions high. There are three possible cases, as shown in Fig. 3.3. Thus, the Slot can be defined by a data structure of type

SKYHAWKE Ex. 1016, page 8

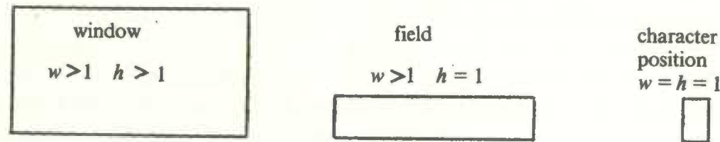

Fig. 3.3. Slots.

```
byte     = 0..255;
SlotType = record
             row,col      : byte;  {starting position}
             width,height : byte;  {size}
           end;
```

The current (that is, the next available) output position on the screen is indicated by a *cursor* symbol — frequently a flashing block or underline character. The commonest way of causing a message to be output in a particular slot is to

- move the current position to the starting (row,col) of the slot;
- use the 'teletype' functions provided by the programming language to output the content, suitably justified, at the new position.

Some programming languages, such as the various BASIC dialects, provide facilities within the language itself for repositioning; others, including PASCAL, do not. However, most microcomputers, dumb VDTs and printers will reposition the cursor in response to the output of a specific string of characters.

The sequence may be a single ASCII character. Most devices will respond to the statement

```
write (chr(12))
```

by 'form feeding', i.e. skipping to a new page on a printer, or clearing the screen and positioning the cursor in the top left corner on a screen. Linefeed (ASCII character 10) and Carriage Return (ASCII character 13) are other examples of such special characters. Normally, a sequence of characters is needed to control positioning; since the first character is usually Escape (ASCII character 27), they are commonly called *Escape sequences*.

The precise sequence for a particular operation varies with the device. A

standard has been produced — the ANSI terminal specification — to which many devices conform. It defines sequences for particular operations, including cursor positioning and clearing areas of the screen; a subset is illustrated in Fig. 3.4. (<ESC> is the ASCII character Escape and the notation <j> means that <j> should be replaced by the relevant value expressed as a character sequence. For example, <ESC> followed by the characters [15A will move the cursor up 15 lines.)

<ESC> [<n> A	move cursor up n lines
<ESC> [<n> B	move cursor down n lines
<ESC> [<n> C	move cursor right n columns
<ESC> [<n> D	move cursor left n columns
<ESC> [<m> ; <n> H	move cursor to column n of line m
<ESC> [1 K	erase from current position to end of line
<ESC> [2 K	erase all of current line

Fig. 3.4. Typical ANSI terminal Escape sequences.

Thus cursor positioning on a device which conforms to the ANSI standard can be accomplished by a procedure of the form:

```
procedure CursorTo(row,col:byte);
begin
write(chr(27),'[',row:1,';',col:1,'H');
end;
```

There is nothing magical about the Escape character. The operating system contains procedures which effect the repositioning and which are invoked in response to these strings. These procedures may also be invoked directly from the application software by suitable low level system calls. Most operating systems will provide the facilities of Fig. 3.5.

procedure CursorTo(row,col:byte)
    positions cursor to (row,col)
procedure CursorAt(var row,col:byte)
    returns the current cursor position (row,col)
procedure SwitchCursor(switch:OffOn)
    causes the cursor symbol to be hidden/displayed
function TestCursor:OffOn
    test whether the cursor symbol hidden/displayed

Fig. 3.5. Cursor control procedures.

SKYHAWKE Ex. 1016, page 10

3.2.3 Output of Text with Particular Format Attributes

Most VDTs provide more display facilities than just the ability to output at any point on the screen. In particular, they often provide a range of *highlighting* features which can be used to enhance the text of a message. Common highlighting features include:

Colour. The text characters (or foreground) can be displayed in a different colour from the background. On some monochrome screens, a similar effect is produced with different shades of the screen's base colour. *Inverse video* is a special case of this feature where the background and foreground colours or shades are swapped.
Blinking. An area of the screen may be made to flash by displaying it alternately in normal and in inverse video. This is commonly used to highlight the cursor position.
Bold Intensity. The ability to support different contrast levels and make an area of foreground appear brighter than surrounding areas.

Highlighting facilities vary from device to device. Some provide only inverse video; others provide a wide range of colour tones and intensity levels. Several provide a number of different fonts (type size and face) for the text characters. In fact, the same device may provide a different range of features depending on the *mode* to which it has been initialised; for example, an IBM-PC may be set to display 40 or 80 columns, colour or monochrome and in various resolutions. In Chapter 6, we discuss guidelines on where and when highlighting features should be used. For the present, we will concentrate on how they are implemented.

In the general case, we may represent the video attributes associated with an output text message by a data structure of type

```
AttributesType = record
                   foreground  : colours;
                   background  : colours;
                   blink       : OffOn;
                   bold        : OffOn;
                   font        : TextStyles;
                   end;
where
  colours      = (black, blue, green ........... );
  OffOn        = (off,on);
  TextStyles   = (roman, ....... );
```

Highlighting facilities may often be invoked by means of escape sequences similar to those for positioning. The range of facilities, and the sequences which invoke them, should be included in the documentation for your particular device. For example, an IBM-PC Escape sequence may be used to switch inverse video on or off with a procedure of the form:

```
procedure inverse(switch:OffOn);
begin
if switch=on then write(chr(27),'[7m')
             else write(chr(27),'[0m');
end;
```

Once a feature has been 'switched on, it remains on until it is 'switched off'. Thus, to output a particular message in inverse at row 10, column 16 requires a sequence of statements of the form:

```
CursorTo(10,16);
inverse(on);
write('this message in inverse at (10,16)');
inverse(off);
```

How does the device know what highlighting features are in force when a particular character is to be displayed? In most microcomputers the screen is *memory-mapped*. For each position on the screen there is a corresponding area of memory which specifies what is to be displayed at that position and in what format; we will refer to this area as the *video map*. In a *character-mapped display*, the smallest addressable position on the screen is a complete character. On an IBM-PC, highlighting features are accommodated by reserving two bytes for each character position on the screen — the first, called the *attribute byte*, specifies how it is to be displayed and the second, called the *content byte*, specifies the character itself — as in Figure 3.6.

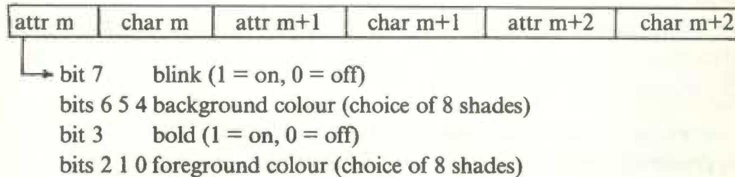

Fig. 3.6. Character video map.

There are 3 bits for both background and foreground colour to correspond to the 3 phosphors (red, green, blue) from which the colours are produced. 'Black' is generated with all 3 bits set to off and 'white' with all 3 bits set to on.

In the example of Figure 3.6, a memory area of 2 × 80 × 25 bytes (approximately 4K) is required for the video map of a standard 80 column by 25 line display. The organisation of the memory map may not correspond directly with the layout of the screen in the way that Fig. 3.6 suggests. Column 1 of row n+1 may not immediately follow column 80 of row n in the map. With a smaller video map, less features can be accommodated; with a much larger map, the system may maintain copies of several physical screens simultaneously or provide a large 'virtual' screen only part of which appears on the physical screen at any one time. We shall see in Chapter 10 that this feature of modern microcomputers can be used to provide an interface technique called *windowing*.

The interpretation of the Escape sequences which set attribute values is supported by routines in a microcomputer's operating systems which write directly into the video map (and hence to the screen). Often system calls are provided which can be invoked directly from the application software; these may provide the programmer with access to more highlighting features than are available with the Escape sequences. For example,

PutChar(content,attribute:byte)

might be an assembler routine which causes the operating system to write the character specified by *content* into the video map at the current cursor position, set the attribute bits to correspond with the value specified by *attributes*, and finally advance to the next position. Most operating systems will support an equivalent to PutChar and a corresponding procedure, GetChar, which returns the content of the current position on the device, and the value of its attributes.

Defining the attributes in terms of actual bit patterns in the video map is both unwieldy and inflexible; it is preferable to specify them in terms of the attribute data structure which we defined earlier. The low level routines can be incorporated into procedures which convert the enumerated types of this data structure into the relevant bit patterns for a particular device. Suitable procedures would take the form

procedure WriteVideoMap(ch:char;attributes:AttributesType)
procedure ReadVideoMap(var ch:char;var attributes:AttributesType)

A string can be displayed with particular attributes by repeated calls to WriteVideoMap, as illustrated by the DisplayString procedure of Fig. 3.7.

```
procedure DisplayString(content:string;
                        attributes:AttributesType);
var size,k : byte;
begin
size:=length(content);
for k:= 1 to size do WriteVideoMap(content[k],attributes);
end;
```

Fig. 3.7. Displaying a message with given attributes.

3.2.4 A Generalised Output Process for Text Messages

We are now in a position to generalise the output of a text message to a device such as a screen. An output message is represented by a variable, *field* (of type FieldType), whose structure is defined by the type declarations of Fig. 3.8.

```
byte            = 0..255;
colours         = (black,blue,green,cyan,red,magenta,yellow,white);
OffOn           = (off,on);
LeftCentreRight = (left,centre,right);

SlotType        = record
                  row           : byte;
                  col           : byte;
                  width         : byte;
                  end;

AttributesType  = record
                  foreground    : colours;
                  background    : colours;
                  blink         : OffOn;
                  bold          : OffOn;
                  justification : LeftCentreRight;
                  end;

FieldType       = record
                  content       : string;
                  slot          : SlotType;
                  attributes    : AttributesType;
                  end;
```

Fig. 3.8. Type declarations for the output message process.

For simplicity, we restrict our discussion in this chapter and the following chapters to the display of a field, i.e. a message which starts at a specified (row,col) character position and extends for a specified number of character positions. In other words, we have omitted 'height' from the SlotType declaration.

Also, for simplicity and because it is a feature which is not available on a number of devices, we have omitted text fonts. It is not difficult to incorporate this facility, although different font sizes may make it tricky to operate in fixed integral character positions.

We have introduced a new attribute — *justification*. The reason for this attribute is merely to make the display of *field.content* more convenient. The number of 'significant' characters in the content string may be less than *field.slot.width*; for example, we may wish to display a short string of black text characters centred within a longer band of blue background. We could assign a string, suitably padded with leading and trailing spaces, to content. Introducing a justification attribute merely saves the designer the trouble of calculating the number of padding spaces required. The string will be displayed automatically with the appropriate number of padding spaces.

We require a mechanism for assigning values to an output field. This is done by the CreateField procedure, which takes the form:

```
procedure CreateField (var field:FieldType;
                       message:string;
                       row,col,width:byte;
                       AttributeString:string);external;
```

The field represented by Fig. 3.9 can be initialised by the statement:

```
CreateField (Figure 3_9, 'My Message Displayed Here',6,10,31,
             'fore=black,back=blue,just=centre');
```

Fig. 3.9.

Let us examine CreateField more closely. Assigning values to content and slot raises no queries but the assignment of attributes deserves some attention. We have already seen that it is desirable to specify the attributes at a conceptual level rather than in terms of how they are implemented. It is tedious to specify a value for each attribute every time we define a field; we will

frequently want the same set of values. More significantly, we will 'bake' this set of attributes into our application software since, if we add a new attribute, we will have to amend all existing software to include the additional value.

Therefore we specify the desired attributes as a string of characters. This string consists of a series of settings separated by commas, and with each setting having the form

keyword = value

where *keyword* identifies a particular attribute (fore, back, blink or bold) and *value* the enumerated type value which the attribute is to take. Any attributes which are not specified explicitly take a default value. In our example, the blink and bold attributes take their default values of 'off' because they are not specified explicitly. The default values for background and foreground are 'black' and 'white' respectively, and the default value for justification is 'left'.

In order to manipulate the field data structures we assume a library of procedures of the form detailed in Fig. 3.10. These procedures utilise the low level routines for positioning and attribute setting described in the preceding sections. Appendix G contains a library of suitable PASCAL procedures. Note that these procedures are not restricted to any particular screen device — device dependence is localised within the low level routines.

The field in Fig. 3.9 can be output to the screen by a statement of the form

DisplayField(Figure3_9);

The content of the field in Fig. 3.9 can be 'cleared' by a statement of the form

ClearField(Figure3_9);

```
procedure ChangeFieldContent(var field:FieldType;
                             NewContent:string);external;
procedure ChangeFieldSlot(var field:FieldType;
                             row,col,width:byte);external;
procedure ChangeFieldAttributes(var field:FieldType;
                             NewAttributes:string);external;
procedure ClearField(field:FieldType);external;
procedure CreateField(var field:FieldType;
                             message:string;
                             row,col,width:byte;
                             AttributeString:string);external;
procedure DisplayField(field:FieldType);external;
procedure HideField(field:FieldType);external;
```

Fig. 3.10. Field manipulation procedures.

This temporarily sets the foreground to the same colour as the background of the field and redisplays it. If the background colour of the field is different from that of the rest of the screen, the area occupied by the field will still be visible but the content will be invisible. This can be used to indicate the location of a message but without displaying the message content.

There are occasions when the dialogue process wishes to draw a user's attention to a message which has already been displayed i.e. to repeat the message with greater emphasis. Rather than creating a new message, this can be done by changing the specification of an existing message and redisplaying it. For example, the statements

```
ChangeFieldAttributes(Figure3_9,'fore'white');
DisplayField(Figure3_9);
```

will redisplay the field in the same position but with white text on a blue background. There are corresponding procedures to change the content or the slot.

Finally, we may wish to hide the area of the screen occupied by the display of a field. This is accomplished by a statement of the form:

```
HideField(Figure3_9);
```

Figure 3.11 illustrates the use of these procedures to display the range of shades available on an IBM-PC colour screen. The program ShowColours outputs a pyramid with shades lightening from black at the top to white at the bottom; each row of the pyramid contains the shade caption in the same colour as the background but in bold intensity.

3.3 Input of a Text Message

3.3.1 Standard Input from the Keyboard

The *keyboard* is the archetypal input mechanism of interactive computing. It can be used for the entry of any text-based input and is appropriate for low to moderate volumes, depending on the proficiency of the user. Where large volumes of input from fairly standard source documents are involved, an automatic data capture device such as an *optical character reader* should be considered; these devices are widely used by public utilities for processing customer account payments and by banks for processing cheques.

```
program ShowColours;
type
{include field Type definitions of Figure 3.8}

var
  OutField       : FieldType;
  row,col,width  : byte;
  k              : byte;
  shade          : array[1..8] of string;

{include field Procedure declarations of Figure 3.10}
procedure ClearScreen;external;

begin
shade[1]:='black' ; shade[2]:='blue' ; shade[3]:='green';
shade[4]:='cyan'  ; shade[5]:='red'  ; shade[6]:='magenta';
shade[7]:='yellow'; shade[8]:='white';
ClearScreen;
CreateField(OutField,'Colours Available',2,1,80,
                     'back=white,fore=black,just=centre');
DisplayField(OutField);
row:=3; col:=35; width:=10;
for k:=1 to 8 do
  begin
  ChangeFieldContent(OutField,shade[k]);
  ChangeFieldAttributes(OutField,
              concat('bold=on,back=',shade[k],'fore=',shade[k]));
  row:=row+1; col:=col-2; width:=width+4;
  ChangeFieldSlot(OutField,row,col,width);
  DisplayField(OutField);
  end;
end. {ShowColours}
```

Fig. 3.11. A colour pyramid.

Other equipment which can be used to substitute or supplement keying includes badge readers (into which you insert your bank service card at the automatic till) and barcode scanners (used to identify products at retail checkout desks). In both cases, their use eliminates the need for the user to enter a long and meaningless string of digits to identify the customer or product code.

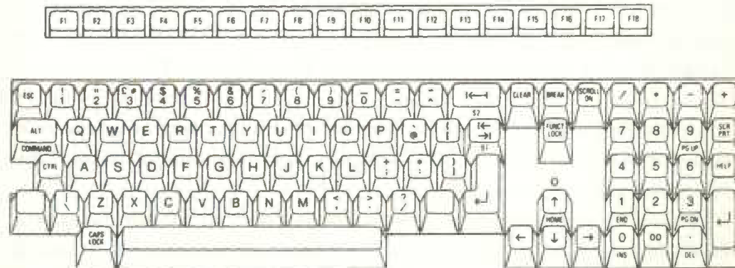

Fig. 3.12. Olivetti M24 keyboard.

In certain systems a full keyboard is unnecessary or undesirable and a customised keyboard may be used, e.g. the numeric keyblock on an automatic bank till. However, unless they are to be produced in large numbers, customised keyboards are expensive. Many improvements on the traditional QWERTY layout, illustrated in Fig. 3.12, have been proposed and shown to reduce the physical strain on the user; thus far it has withstood all competitors.

The basic mechanism by which input is received by a program is the same for all keyboards. Most high level languages provide a procedure to read input delimited by an end of line marker such as Carriage Return. Thus in PASCAL, the statement

```
read (x)
```

reads a value from the keyboard and assigns it to variable **x**.

The use of a standard input facility, like READ, has a number of implications for the interface. The processing which occurs to effect the input is as follows:

repeat
- *wait* for input to become available;
- *interpret* any control sequences for editing (such as Backspace or Delete) or screen positioning;
- *echo* the edited/interpreted input to an attached screen;

until end of input is indicated by the user pressing the Carriage Return key.

In a language such as PASCAL, the input will then be checked for type consistency with the input variable and *converted* as appropriate, or will cause an error.

For many text-based applications, the processing described above may be exactly what is required. However, it does have limitations. The application may not want the dialogue to wait for input, but rather to examine the keyboard to see whether or not a key has been pressed. This is necessary where the system repeatedly carries out a process until the user presses a key to terminate it. A common example occurs when a source file is listed on the screen. Lines from the file scroll up the screen until the user causes a pause, typically by pressing control-S.

It may be undesirable to echo what the user types onto the screen. Obviously password protection would be of little use if the password were to be displayed whenever it was entered.

The application may interpret editing keys in a non-standard manner.

During execution of a PASCAL read instruction, the Backspace key is interpreted as a *destructive backspace*, i.e. the previous character is overwritten. With a form displayed on the screen, the dialogue may want to interpret the Backspace key as 'position the cursor at the preceding field'.

If the dialogue is expecting a single character reply, the user should not be required to press the return key; the input should terminate when one character has been entered.

3.3.2 Character-by-Character Input

To support these variations, many languages allow *character-by-character* input. Even if the language (as with PASCAL) does not include it, many operating systems provide the facility via a low level routine which can be linked with the language's procedures. The essentials of such input are that:

The application software can interrogate the keyboard to see if a key has been pressed. If it has, the character code corresponding to the key will be returned; otherwise an indicator such as ASCII NUL (**ascii 0**) will be returned.

The majority of input codes will not be interpreted to see if they represent control sequences (it may not be possible to trap some Resets). In particular, a Backspace will appear as the ASCII character BS (**ascii 8**).

The input will not be echoed automatically onto the screen.

This facility gives the designer the ability to control the way in which input is received from the keyboard. A program can wait for input by repeatedly scanning the keyboard until a non-null character is obtained. Input can be echoed by displaying characters as they are received; in fact, numeric input can be echoed right-justified, and alphabetic input can be echoed left-justified.

Since control sequences are not interpreted by the operating system, the dialogue process can interpret any character, or sequence of characters, in any way. The corollary of this freedom is that it must undertake any editing functions required.

A final point to note is that this mechanism only supports input of type Char. Although this may seem a disadvantage, a little thought should suggest that all text input (including that of numbers) should be as characters which are then converted explicitly by the application software. No system should crash because a keying error caused a type mismatch in a PASCAL program!

Handling character-by-character input requires the existence of an external procedure of the form

SKYHAWKE Ex. 1016, page 20

function GetKeyChar(WaitSwitch:WaitType;EchoSwitch:EchoType):byte;

where WaitType = (Wait,NoWait) and EchoType = (Echo,NoEcho) and the byte value returned contains the ASCII code of the character corresponding to the key.

3.3.3 Handling 'Special' Keys

Most keyboards contain a number of 'special' keys in addition to the alphabetics and the numeric digits. These 'special' keys include editing keys such as Backspace, cursor control keys and function keys. If GetKeyChar returns the corresponding ASCII code when an alphabetic or numeric is pressed, what does it return if a special key such as the cursor ↑ or function key F1 is pressed?

Keys such as Backspace and CarriageReturn also return a single ASCII code. On some devices, all the keys operate in this manner; that is you can test for function key F1 by a statement of the form

if key = n for some $0 < n < 256$

Unfortunately, not all devices are so helpful. Many terminals and microcomputers (such as the IBM-PC) return a sequence of several characters for the depression of a single function or cursor control key. Often this sequence starts with the ASCII NUL character which is also used to indicate that no key has been pressed; a function like GetKeyChar will not handle special keys on these devices.

There is a solution to this problem. Each key on the keyboard (even ShiftLock) has a unique numeric code, a *scan code* associated with it. The scan codes for the keyboard illustrated in Fig. 3.12 are shown in Fig. 3.13. When a key is depressed, the corresponding scan code is generated. Some keys (Shift, Control, CapsLock, etc.) do not produce an input 'character'. They

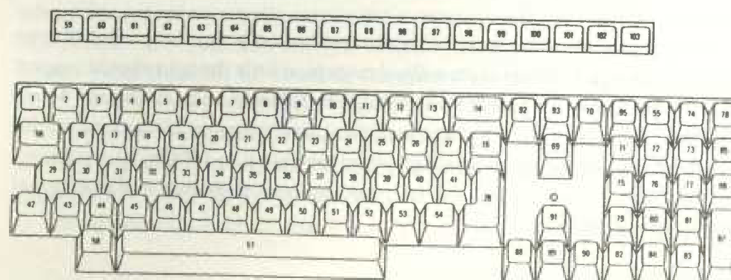

Fig. 3.13. Olivetti M24 scan codes.

SKYHAWKE Ex. 1016, page 21

have no meaning except when combined with a 'character' key and merely set a KeyBoard State flag. If the 'c' key is depressed, the keyboard handler translates it as 'C' or 'c' or 'control-C' and generates the corresponding ASCII code by interpreting the scan code in light of the Keyboard State flags. A sequence of ASCII codes for a function key or cursor control key is generated in a similar manner.

Many operating systems provide entry points which enable low level routines to access the scan codes rather than just the translated ASCII codes. Consider an external procedure of the form

procedure GetKeyScan(WaitSwitch:WaitType;EchoSwitch:EchoType;
var scan,ascii:byte); **external;**

This operates in a similar way to GetKeyChar except that it returns values for both the scan code and an ASCII interpretation of this code. The scan code will be non-zero if and only if a key has been pressed. For a normal key, the scan code will be interpretable as an ASCII value which will be returned in the variable ASCII; for special keys, no such interpretation will be possible and so the value of ASCII will be zero.

By associating with a single key an action which might otherwise require several keystrokes, the amount of keying required from the user can be minimised. The existence of a procedure like GetKeyScan enables the dialogue to interpret function and cursor keys in any way it wants. Since the normal 'character' keys produce ASCII codes in the range 0..127, one simple way of doing this is to assign special keys a number in the range 128..255, and to translate their scan codes on input as in the general keyboard input procedure, GetAnyKey, illustrated in Fig. 3.14. Appendix F contains a list of the keycodes used in this book to represent special keys.

There is a further refinement which we can incorporate into the process. Suppose that the dialogue requests the user to enter a customer code which is all numeric. If the user keys a character other than a digit from 0 to 9, it is obvious that a typing error has occurred. The input process could accept the field with this error and let the dialogue process validate it, reject it, output an

```
function GetAnyKey(WaitSwitch:WaitType;EchoSwitch:EchoType):byte;
var  scan,ascii : byte;
begin
GetKeyScan(WaitSwitch,EchoSwitch,scan,ascii);
if (scan<>0) and (ascii=0) then ascii:=scan+128;
GetAnyKey:=ascii;
end; {GetAnyKey}
```

Fig. 3.14. Character-by-character input of any key.

SKYHAWKE Ex. 1016, page 22

error message and request re-input. This is a long-winded approach to such a trivial mistake; it would be easier if the input process simply ignored the miskeyed character.

This facility can be incorporated by specifying a *filter* which defines the set of keys to which the input process is sensitive. Any characters entered which are not in this filter are ignored; a suitable procedure, GetFilterKey, is illustrated in Fig. 3.15.

```
type
   WaitType  = (wait,NoWait);
   EchoType  = (echo,NoEcho);
   SetOfByte = set of byte;

function GetFilterKey(WaitSwitch:WaitType;
                      EchoSwitch:EchoType;
                      filter    :SetOfByte):byte;
var  scan,ascii : byte;
begin
ascii:=0;
repeat
   GetKeyScan(WaitSwitch,EchoSwitch,scan,ascii);
   if (scan<>0) and (ascii=0) then ascii:=scan+128;
   if not (ascii in filter) then ascii:=0;
until (ascii<>0) or (WaitSwitch=NoWait);
GetFilterKey:=ascii;
end; {GetFilterKey}
```

Fig. 3.15. Character-by-character input via a filter.

3.3.4 A General Input Process for Text Messages

In the previous sections, we were concerned with input only at the level of individual characters. In practice, a general input text message will consist of a string of characters up to a given length. This introduces further requirements.

First, the process requires a variable in which to return the content of the input message; by definition, this will be of type string. If the input is to be echoed to the screen, there must be a definition of the slot on the screen where this content will be echoed and a definition of the format (attributes) in which it is to be echoed. This appears a remarkably similar requirement to that for an output text message, and indeed an input field has the same type declaration:

var InputField : FieldType

where

```
FieldType = record
              content    : string;
              slot       : SlotType;
              attributes : AttributesType;
              end;
```

Note that the slot defines not only where the input is echoed. By specifying a width, it also defines a maximum length for the input.

The user may make a keying mistake; a *rubout* character must be defined to permit editing of the input. If a user can abort the system at any point, a *RequestAbort* character must also be defined. An *EchoSwitch* is required to specify whether the input is to be echoed. A *RequestHelp* character is needed to permit the user to request help.

The input process must be able to determine when to terminate. As we discussed earlier this can be signalled either by the input of a special terminator character or by the input of a fixed number of characters. An *AcceptField* character can be defined to cater for the first case; the process will continue until this character is input, regardless of the number of characters which have been entered. If more characters are entered than the slot can contain, any excess will be ignored. In the second case, there is no explicit terminator character and so AcceptField will be defined as null; the process will continue until exactly slot.width characters have been entered.

The number of such 'switches' which are required to control the input process would result in a very unwieldy parameter list if they were all passed as individual parameters. We will collect them all into a single data structure which we will refer to as a *Dialogue Information Block* (DIB). Because this structure contains settings which control the input process we will call it a control DIB.

The complete structure of the control DIB is illustrated in Fig. 3.16. It contains a number of other elements in addition to those which we have already described. The purpose of these additional elements will be described as they are required.

Combining all these requirements results in a procedure of the form:

```
procedure ReadField (var field:FieldType;
                     DataSet:SetOfByte;
                     var ControlDIB:ControlDIBtype)
```

where DataSet is a filter defining the 'keys' to which the process is sensitive.

SKYHAWKE Ex. 1016, page 24

```
ControlDIBtype = record
                 ControlBuffer        : byte;
                 {input of text message}
                 rubout               : byte;
                 EchoSwitch           : OffOn;
                 AcceptField          : byte;
                 {standard controls}
                 RequestAbort         : byte;
                 RequestHelp          : byte;
                 {picking & pointing controls}
                 reserved1            : byte;
                 reserved2            : byte;
                 reserved3            : byte;
                 {form positioning controls}
                 reserved4            : byte;
                 reserved5            : byte;
                 reserved6            : byte;
                 reserved7            : SetOfByte;
```

Fig. 3.16. The Control DIB data structure.

To accept the input of a numeric field of up to 6 digits terminated with a CarriageReturn and to echo right-justified in inverse video starting at (20,10) requires a program fragment of the form:

```
var ControlDIB  : ControlDIBtype;
    InField     : FieldType;
    DataSet     : SetOfByte;

with ControlDIB do
   begin
   rubout:=BS;              {keycode 8}
   EchoSwitch:=on;
   AcceptField:=CR;         {keycode 13}
   RequestAbort:=ESC;       {keycode 27}
   end;

{display blank slot for answer in inverse video}
CreateField(InField,'',20,10,6,'fore=black,back=white,just=right');
DisplayField(InField);

DataSet:=[ord('0')..ord('9')]; {filter only allows digits 0-9}
ReadField(InField,DataSet,ControlDIB);
```

To accept an input of exactly 6 alphabetic characters without echo behind a mask of asterisks starting at (20,10) requires a program fragment of the form:

```
with ControlDIB do
   begin
   rubout:=BS;              {keycode 8}
   EchoSwitch:=off;
   AcceptField:=null;       {keycode 0, no explicit terminator}
   RequestAbort:=ESC;       {keycode 27}
   end;
```

```
{display slot for answer filled with asterisks}
CreateField(InField,'******',20,10,6,'');
DisplayField(InField);
DataSet:=[ord('a')..ord('z'),ord('A')..ord('Z')];
ReadField(InField,DataSet,ControlDIB);
```

In both cases, the user's input will be returned in InField.content. Although in the second case the input characters are not echoed, the cursor will still track the input position along the asterisks. Of course, the user may decide not to enter a text message but may choose instead to press the RequestAbort key. ReadField will terminate immediately but this is unlikely to be a sufficient response by the system; the dialogue needs to know that the user has aborted the process. The ControlBuffer in ControlDIB provides the means for this; when ReadField returns, ControlBuffer will contain the keycode of Request-Abort. We shall see later that there are other occasions when this ControlBuffer is used for communicating 'control' inputs to the dialogue.

A suitable ReadField procedure is detailed in Appendix G. It makes use of the low level routines which were discussed in the previous section.

3.4 Positioning, Pointing and Picking

3.4.1 Targets

In Chapter 2, we saw that an Input Control message typically involves selection from a limited set of options; an Input Data message may also involve such a selection. It can be accomplished by keying a text message which identifies the option desired. It can also be accomplished if the list is reasonably small, by the system displaying the choices available and the user 'pointing' at the one required.

The alternatives are represented by a list of targets. These targets may be individual characters; in editing a piece of text, the selections possible are the character positions on the screen. They may be a set of character strings; such as a list of task processes which can be executed, or a list of file names which may be opened. In the more general case, they may be a series of targets represented pictorially on the screen, as in Fig. 3.17.

SKYHAWKE Ex. 1016, page 26

indicate that he has reached the target he wants by entering the character specified as *AcceptTarget*. A means of aborting the mechanism without making a selection is also desirable.

Where are these characters to be defined? In Section 3.3.4, we introduced the concept of a Dialogue Information Block (the ControlDIB) which contained the various controls for the input process for a text string. PointNextTarget, PointPriorTarget and AcceptTarget are corresponding controls for the Relative Pointing process; these are also stored in the controlDIB data structure as shown in Fig. 3.18.

The procedure in Fig. 3.19 illustrates the use of this mechanism; it returns in *CurrentTarget* the ordinal number of the target picked. The target at which the user is currently pointing is highlighted with the attributes specified in the corresponding input parameter. SwitchCursor is an external routine which switches the normal screen cursor on or off. Why is this necessary? (The reader may note that the reassignments of CurrentTarget in response to the input of PointNextTarget or PointPriorTarget could be coded more elegantly, if less transparently, using the MOD function to cycle CurrentTarget through the values 1..NumberOfTargets.)

3.4.3 Absolute Pointing

Cursor control keys can be used to move, one character at a time, to any position on the screen. However a series of crablike vertical and horizontal

```
ControlDIBtype = record
                 ControlBuffer        : byte;
                 {input of text message}
                 rubout               : byte;
                 EchoSwitch           : OffOn;
                 AcceptField          : byte;
                 {standard controls}
                 RequestAbort         : byte;
                 RequestHelp          : byte;
                 {picking & pointing controls}
                 PointNextTarget      : byte;
                 PointPriorTarget     : byte;
                 AcceptTarget         : byte;
                 {form positioning controls}
                 reserved4            : byte;
                 reserved5            : byte;
                 reserved6            : byte;
                 reserved7            : SetOfByte;
```

Fig. 3.18. The Control DIB data structure with Pointing Controls.

```
procedure RelativePick(TargetList:TargetListType;
                       NumberOfTargets:byte;
                       highlight:AttributesType;
                       var ControlDIB:ControlDIBtype;
                       var CurrentTarget:byte);
var
   complete : boolean;
   filter   : SetOfByte;
begin
with ControlDIB do
   begin
   filter:=[RequestAbort,RequestHelp,
            PointNextTarget,PointPriorTarget,AcceptTarget];
   SwitchCursor(off);
   if (CurrentTarget<1) or (CurrentTarget>NumberOfTargets) then
      CurrentTarget:=1;
   HighlightField(TargetList[CurrentTarget],highlight);
   complete:=false;
   repeat
      ControlBuffer:=GetFilterKey(Wait,NoEcho,filter);
      {ControlBuffer cannot be zero}
      if ControlBuffer=RequestAbort then
         complete:=true
      else
      if (ControlBuffer=PointNextTarget)
      or (ControlBuffer=PointPriorTarget) then
         begin
         DisplayField(TargetList[CurrentTarget]); {turn off old highlight}
         if ControlBuffer=PointNextTarget then
            if CurrentTarget=NumberOfTargets then CurrentTarget:=1
                                 else CurrentTarget:=CurrentTarget+1
         else
            if CurrentTarget=1 then CurrentTarget:=NumberOfTargets
                               else CurrentTarget:=CurrentTarget-1;
         HighlightField(TargetList[CurrentTarget],highlight);
         ControlBuffer:=0;  {because it has been actioned}
         if AcceptTarget=0  then complete:=true;
         end
      else
      if ControlBuffer=AcceptTarget then
         begin
         complete:=true;
         ControlBuffer:=0;  {because it has been actioned}
         end
      else
         {terminated by a higher level control input}
         complete:=true;
   until complete;
   SwitchCursor(on);
   end;
end; {RelativePick}
```

Fig. 3.19. Scrolling round a set of targets.

motions is hardly natural or speedy. Free pointing anywhere on the screen really requires a specialist pointing device. The operating system will contain low level routines to support the operation of the device; a PASCAL program

will typically access these routines via calls to external procedures. The format in which these routines report device activity varies from device to device. However, it should be possible to produce a generalised procedure of the form:

procedure ReadPointer(var row,col:byte;var action:byte)

action specifies what activity, if any, has occurred; for example, we need to be able to tell if the user has picked a target, requested an abort and so forth. The position of the device is specified by (row,col); most pointing devices report position in terms of pixel co-ordinates but these can easily be converted to row and column values for use in text-based applications.

In order to determine which target, if any, is being pointed at, these row and column co-ordinates must be matched against the co-ordinates of the target areas. A target is pointed at if the character position (row,col) lies with the target slot. The point and pick procedure must try all targets in the list, as illustrated in Fig. 3.20, until a match is found; if the list is exhausted before a match is found, then the user is not pointing at any target.

The mouse (Fig. 3.21) is the pointing device in vogue. Although developed more than a decade ago, recent dramatic reductions in cost have resulted in its incorporation in most modern workstations. It consists of a box about the size of a cigarette packet attached to the keyboard or processor; set into the top are a number (1 to 4) of buttons which operate rather like function keys. As the mouse is moved in the palm of the hand over a convenient surface (a desktop or a tablet), its position is tracked relative to some preset origin using

```
function MatchPosition(row,col:byte;
                       TargetList:TargetListType;
                       NumberOfTargets:byte):byte
var k,match : byte;
begin
match:=0;
k:=1;
while (k<=NumberOfTargets) and (match=0) do
   if  (row=TargetList[k].slot.row)
   and (col>=TargetList[k].slot.col)
   and (col< TargetList[k].slot.col+TargetList[k].slot.width) then
      match:=k
   else
      k:=k+1;
MatchPosition:=match;
end; {MatchPosition}
```

Fig. 3.20. Matching a target area.

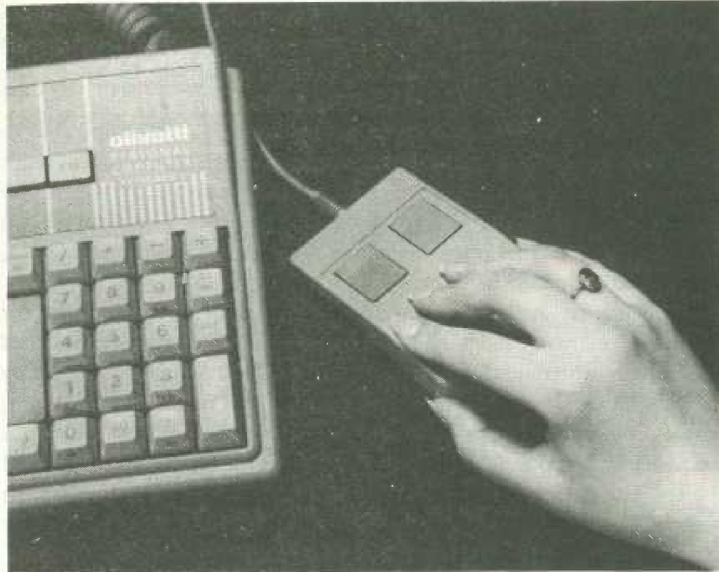

Fig. 3.21. An Olivetti mouse.

one of two methods. The cheaper and more common mechanical mouse rolls on a ball bearing; the motion of the ball bearing is transferred to two rollers which are perpendicular to each other and which track motion in the x and y planes. The optical mouse which is common in graphical applications uses an optical crosshair over a special tablet upon which a very precise grid has been etched.

It was mentioned in Section 3.2.3 that the output features provided on some screens depend on the *mode* to which they have been initialised. Similarly, many mice exhibit different operating characteristics depending on the mode to which they have been *configured.* This configuration involves supplying values for a variety of parameters which control the mouse operation; for example, one parameter will specify the sensitivity of the mouse i.e. the scale of physical movement necessary to effect one unit of movement on the screen.

A common facility used with text-based dialogues is *keyboard emulation.* Movements of the mouse and presses and releases of the buttons are converted to associated key character strings. Typically, a mouse movement

to the left is converted to the character generated by the cursor left key, a press of Button 1 is converted to a CarriageReturn character, and so forth. The key strings which are 'emulated' for each mouse action can be set by the configuration routines.

The dialogue has no need to know of the existence of a mouse operating in keyboard-emulation mode. Use of the mouse imposes no special software requirements since its inputs will be processed correctly by any of the keyboard routines which we have discussed previously. Therefore we will concentrate on the case where the mouse operates as a independent device, often called *real mouse* mode.

There are no intrinsic procedures in most high level languages for processing real mouse input i.e. there is nothing that corresponds to the Read and Readln procedures for the keyboard. All input/output from auxiliary devices like the mouse must be handled via low level routines called *drivers*; in fact, the routines like WriteVideoMap and GetKeyScan which we discussed in Sections 3.2 and 3.3 are examples of Screen and Keyboard drivers. The precise implementation of the Mouse driver depends on the particular mouse. We will consider the types of facility provided by the driver of a typical mechanical mouse. These are listed in Fig. 3.22.

Movement of the mouse is tracked on the screen by a mouse cursor symbol. A number of different cursor symbols may be supported — a block, a character attribute setting or a symbol such as an arrowhead or sightmark. The cursor symbol can be set by a statement of the form:

SetMouseCursor(inverse);

It may be desirable to turn the cursor off; for example, when a particular target is highlighted, a flashing cursor is an unnecessary distraction. This requires a statement of the form:

```
procedure SetMouseCursor(cursor:CursorType);external;
procedure SwitchMouseCursor(switch:OffOn);external;
procedure SetMouseLimits(limits:rectangle);external;
procedure MouseTo(row,col:byte);external;
procedure MouseAt(var row,col:byte);external;
procedure ReadPress(button:ButtonType;
                    var report:ReportType);external;
procedure ReadRelease(button:ButtonType;
                      var report:ReportType);external;
```

Fig. 3.22. Typical real mouse facilities.

```
SwitchMouseCursor(off);
```

The area of the screen over which the mouse can travel can be restricted, for example, to constrain the user to positioning within a given rectangular area such as a list of options. This requires a statement of the form:

```
SetMouseLimits(TargetListArea);
```

Just as the normal screen cursor can be initialised to a given position, the mouse cursor can be set to any given position by a statement of the form:

```
MouseTo(10,25);
```

On many systems, the designer must take care to avoid conflicts between the mouse cursor and the normal text cursor. The system may well treat these as different entities. In particular, the fact that the mouse cursor is displayed at a particular position on the screen does not necessarily imply that an output to the screen will occur at this position; output will take place at the position pointed at by the text cursor.

Various mouse actions can be reported. The application software can determine the current mouse position and whether the buttons are currently up or down. As well as these absolute reports, the mouse can provide relative reports of activity since it was last interrogated, such as the relative movement or the number of times a button has been pressed or released and the absolute position at which this last occurred. The second of these is useful for checking whether the user has 'clicked' (i.e. pressed and released a button) in a particular target area.

A report on the current mouse position (row,col) can be read with a statement of the form:

```
MouseAt(row,col);
```

Reading 'clicks' from the mouse requires statements of the form:

```
ReadPress(LeftButton,report);
ReadRelease(RightButton,report);
```

where *report* is a variable of type

```
ReportType = record
             count : byte;  {count of presses or releases}
             row   : byte;  {row of last press or release}
             col   : byte;  {col of last press or release}
             end;
```

SKYHAWKE Ex. 1016, page 32

and returns the number of presses (releases) of the specified button and the position at which the last one occurred. Button is an enumerated value from

ButtonType = (LeftButton,RightButton)

The way in which a button is pressed can also be coded. A common convention is that a single 'click' selects/deselects a target, holding the button down drags a target across the screen, and a 'double click' (i.e. two clicks in quick succession) indicates some operation on the target, such as opening a file.

Alternatives, which operate in a similar manner to the mouse, include the *tracker ball* and the *joystick*. The *touchscreen* and the *lightpen* both differ from the mouse in that a user can point and pick in a single operation. The dialogue process should not be concerned with which of these particular devices is being used; therefore we need to define an abstraction for absolute pointing which generalises to all of these. We will illustrate this by considering the process for a mouse with two buttons which are 'single clicked'.

The ReadPointer abstraction must provide sufficient information to decide which target, if any, is currently pointed at; this can be accomplished if its implementation returns the (row,col) position. It must also indicate whether a target has been picked or an abort has been requested. For relative pointing, we defined a particular keycode for each of these in ControlDIB; the mouse input routine, which knows about buttons, implements this by converting clicks (releases of a button) to keycodes which the dialogue knows about. Fig. 3.23 illustrates the mechanism.

Similar generalisations are required for the cursor control procedures of Fig. 3.22. The dialogue requires an abstraction for any pointing device with the same parameter format; for example:

MouseTo(row,col:byte) => PointerTo(row,col:byte)

These can be incorporated into a generalised procedure for absolute picking as illustrated in Fig. 3.24. The boolean function equal(a,b:byte):boolean returns the value 'true' if and only if (a=b) and (a<>0). It is required to prevent a zero result in the ControlBuffer matching an undefined (and hence zero value) control variable (for example RequestAbort) in ControlDIB.

Point and pick devices can be used very sucessfully provided the user is not continually having to alternate between the device and a keyboard. The mouse and the touchscreen in particular have both proved very acceptable to and easily used by a wide range of users. The touchscreen has the disadvantage at present of being very expensive relative to the other

```
procedure ReadPointer(var row,col:byte;var action:byte); {mouse}
type ReportType = record
                  count : byte;
                  row   : byte;
                  col   : byte;
                  end;
var  report : ReportType;
{include Mouse procedure declarations of Figure 3.22}

begin
ReadRelease(RightButton,report);
if report.count>0 then      {right button release = abort}
   begin
   row:=report.row;
   col:=report.col;
   action:=ESC;             {keycode 27}
   end
else
   begin
   ReadRelease(LeftButton,report);
   if report.count>0 then   {left button release = pick}
      begin
      row:=report.row;
      col:=report.col;
      action:=CR;           {keycode 13}
      end
   else
      begin
      MouseAt(row,col);     {no clicks but check position}
      action:=0;            {keycode 0}
      end;
   end;
end; {ReadPointer for a Mouse}
```

Fig. 3.23. Checking for pointing device activity.

mechanisms, but with the advantage of not impeding keyboard use in any way.

3.5 Input and Output of Graphical Messages

We defined a graphical message as one in which the information exchanged must be described at the 'bit' level rather than at the 'character' level. It is not our intention to discuss graphics in this book but we will mention a few of the factors which are similar to text input and output.

A screen memory map need not store the screen contents as characters; it may hold them instead as *pixel* (short for picture element) information. A

```
procedure AbsolutePick(TargetList:TargetListType;
                       NumberOfTargets:byte;
                       highlight:AttributesType;
                       var ControlDIB:ControlDIBtype;
                       var CurrentTarget:byte);
var
   complete    : boolean;
   PriorTarget : byte;
   row,col     : byte;

begin

with ControlDIB do
   begin
   SwitchCursor(off); SwitchPointerCursor(on);
   if (CurrentTarget>=1) and (CurrentTarget<=NumberOfTargets) then
      begin
      PointerTo(TargetList[CurrentTarget].slot.row,
                TargetList[CurrentTarget].slot.col);
      HighlightField(TargetList[CurrentTarget],highlight);
      end;
   complete:=false;
   repeat
      PriorTarget:=CurrentTarget;
      ReadPointer(row,col,ControlBuffer);
      if equal(ControlBuffer,RequestAbort) then
         complete:=true
      else
         begin
         (ControlBuffer may be zero)
         CurrentTarget:=MatchPosition(row,col,TargetList,NumberOfTargets);
         if (CurrentTarget<>0) and (AcceptTarget=0) then
            complete:=true
         else
         if CurrentTarget<>PriorTarget then
            begin
            if PriorTarget<>0 then DisplayField(TargetList[PriorTarget]);
            if CurrentTarget<>0 then HighlightField(TargetList[CurrentTarget
                                                    ,highlight);
            end
         else
         if  (ControlBuffer<>0)
         and (ControlBuffer in [AcceptTarget,RequestHelp]) then
            complete:=true;
         end;
   until complete;
   SwitchPointerCursor(off); SwitchCursor(on);
   end;
end; (AbsolutePick)
```

Fig. 3.24. Absolute pointing at a set of targets.

pixel is the smallest addressable element of the screen. Pixels might be considered as the dots which are combined to create the normal character set. On a monochrome screen, one bit is required to specify each pixel (it is either on or off) and thus such displays are often referred to as *bit mapped displays*;

on a colour screen, a number of bits representing the basic phosphor colours are required for each pixel.

Analogous to the WriteVideoMap and ReadVideoMap procedures for characters discussed in Section 3.2.3, the operating system may include WriteDot and ReadDot procedures which write or read a pixel value at the current position. Frequently, these basic facilities are incorporated into procedures which support the drawing of basic shapes such as a straight line betweeen two points, a circle or a rectangle. Graphic input may be obtained via a mouse. Another common input device is the tablet which consists of a flat slab used in conjuction with a pen-like stylus. The position of the stylus can be detected by a variety of techniques; a common mechanism utilises the pressure of the stylus to vary the electrical properties of a membrane coating on the tablet. A pressure sensitive tablet can be used both for freehand drawing and for tracing.

The tablet can also be used for point and pick operation. Rather than display the targets on the screen, they are printed onto a transparent overlay which rests on the pad. Both because of the smaller typography which is possible and because of the greater precision of the stylus, targets can be smaller and more cluttered than with other devices.

3.6 Summary

The dialogue process treats input and output messages at a logical level, i.e. in terms of the functions they fulfill.

The input/output processes deal with the raw material of the messages. They are not concerned with their function only with their format which, for text-based dialogues, can be classified into

the output of a text message
the input of a text message
point and pick input

The intrinsic procedures of most high level languages only support 'teletype' operation. Low level routines provide many facilities which can enhance the quality of the dialogue but introduce device-dependence. To facilitate *portability* of the system, it is important to localise this dependence.

The input/output processes are split into two levels. The Dialogue Process

calls the higher level which represents *abstractions* concerned with what is to be effected rather than how it is implemented.

This higher level calls lower level physical device *drivers* which implement the abstractions on a particular device. Implementation for a different device involves the 'linking' of a different library of driver routines.

An input process must be able to distinguish inputs which represent data from those which control the input process itself; an example of a control input is the rubout key. A data structure, which we have called ControlDIB, parameterises the control inputs of the abstractions; any given dialogue can assign its own particular keycodes to these parameters.

This layering of the processes into different libraries of routines is described further in Appendix D. The library of input/output handling abstractions developed in the chapter are described in more detail in Appendix G and the driver routines for the screen, keyboard and mouse are listed in Appendix F.

Discussion Exercises

D1. An application which was developed for use on a device with a colour screen is to be 'ported' to a device with a monochrome screen which only supports inverse video. The dialogue abstractions assume that colour and blink are available. How should the driver for the monochrome screen interpret different colour settings and blink? (Consider examples such as a white foreground on a blue background and a black foreground on a green background.)

D2. Some applications use sound as a highlighting feature; when a field is displayed the associated sound is produced, for example to indicate that the dialogue expects input. Most devices provide a mechanism to 'ring the bell' and several support tone generators which can produce quite elaborate tunes. What extensions to the abstractions described in the chapter would be necessary to incorporate this facility?

D3. In question D2 of Chapter 2 you were asked to investigate different types of software package in terms of their basic dialogue structure and grammar. Re-examine these packages in terms of the input/output processes they support; for example, what devices are supported, whether they utilise pointing, what video attributes are used.

What benefits or disadvantages do these features bring to interface in each example? Why? Most packages must be *installed* for the particular devices

which are used. Determine what type of information about the devices must be specified during installation of these packages.

Programming Exercises

P1. Determine how the cursor control and attribute settings discussed in Section 3.2 can be implemented on the device you are using. In many cases (such as IBM compatibles), this can be done by writing suitable control sequences. Implement the library of screen driver routines listed in Appendix F.

P2. Implement the library of keyboard driver routines discussed in Section 3.3 and listed in Appendix F. On many devices, the keycodes which are produced by a given key (such as the function keys) can be set via particular Escape sequences. This can be considered as setting the *mode* for the keyboard in the same way that the operating mode of a screen or a mouse must be initialised.

P3. What field definitions are required to produce the following 'boxed' list of options? The text is white on a blue background. The character set of most microcomputers includes line segment symbols which can be used to 'draw the box' in text mode.

```
(8,21)
        +--------+
        | happy  |
        | dozy   |
        | lazy   |
        | sleepy |
        | grumpy |
        +--------+
                  (14,32)
```

Use the procedures described in the chapter to display the box on the screen and to scroll around it.

P4. Repeat the previous question but instead of scrolling around the screen use the ReadField procedure to request the user to type one of the letters h,d,l,s or g. This input will not be echoed but the corresponding adjective in the list will be highlighted by 'inversing' it.

P5. Develop a

procedure SaveSlot(slot:SlotType:var buffer:BufferType);

where BufferType = array[1..80] of BufferEntry
and BufferEntry = record
ch : char;
attributes : AttributesType;
end;

which copies the contents of the video map corresponding to *slot* into the variable *buffer*.

P6. Although absolute pointing using cursor keys is cumbersome, it is perfectly possible to point absolutely with a mouse which is operating in keyboard emulation mode. To the system, this is equivalent to pressing the cursor keys. Figure 3.23 illustrated the implementation of ReadPointer for a mouse in real-mouse mode. Implement a corresponding driver for cursor key input.

P7. Display a keyboard layout on the screen as illustrated below. Each key will be represented as an individual field and initially the key captions will be blank. Implement a procedure which will scan the keyboard for an input keystroke and will display a suitable caption on the relevant key. Thus for an input of

c = c will appear on the C key
C = C will appear on the C key and RS/LS on the shift keys
Ctl+c = ^C will appear on the C key and Ctl on the control key

Bsp = Backspace CR = Carriage return Ctl = Control
Esc = Escape LS = Left Shift RS = Right Shift

Further Reading

Biggerstaffe T. (1986) *System Software Tools* (chapter 3), Prentice Hall.

Cakir A. *et al.* (1980) *The VDT Manual*, Wiley.

Carroll A.B. (1984) 'Three Types of Touch Technology Simplify Man–Machine Interface', *Computer Tech. Rev.*, Winter.

Lopiccola P. (1983) 'Meet the Mouse', *Popular Comp.*, **2**, 5.
Montgomery E.B. (1982) 'Bringing Manual Input into the 20th Century', *IEEE Computer*, **15**, 3.
Pfaff G. *et al.* (1982) 'Constructing User Interfaces Based on Logical Input Devices', *IEEE Computer*, **15** 11.
MicroSoft Corp. (1984) MS-DOS Programmer's Reference.
Olivetti (1984) M24: Hardware Architecture and Function.

SKYHAWKE Ex. 1016, page 40

Chapter 5

Dialogue structures – forms, commands and hybrids

5.1 Introduction

In Chapter 4 we examined the characteristics of the Question and Answer and of the menu structures. Both request a single answer to a single question and we saw that the same abstraction can be used to define the input process in either structure. In this chapter, we consider two structures which request a series of answers and will see that this same abstraction applies to the processing of each individual answer in the series.

The four structures represent a broad classification of a single step in a dialogue. Each is suited to a particular class of user or type of input message. However, the dialogues for most applications must handle a variety of input message types and different levels of familiarity; no one dialogue structure is suitable for the whole of the dialogue. We examine how these basic structures may be combined to cater for differing requirements in different areas of the system. We will refer to these combinations as *hybrid* structures.

5.2 The Form Filling Structure

5.2.1 Features

Field Definition

Content: [**My Message Displayed Here**]

Slot:
row [**6**] column [ ] width [ ]

Attributes:
Foreground [ ] Background [ ]
Bold (y/n) [ ] Blink(y/n) [ ]

F1 = Accept Form F2 =help ESC = abort

Fig. 5.1. A form dialogue.

One way of obtaining information from other people is to ask them questions and listen to their replies, an approach reflected in the Question and Answer structure. Another approach is to have them fill in the information on a form, such as the one shown in Fig. 5.1.

Forms are widely used for ordering goods, making reservations or payments, completing insurance proposals, as questionnaires and so forth. The clerical procedures in most companies are based on standard forms such as invoices, sales orders and purchase orders; in fact forms are generally used where the recording of an activity (*transaction*) requires the entry of a fairly standard set of data items.

The form dialogue structure is based on an analogy with this way of collecting information. Unlike the Q&A structure which presents questions one at a time, the user is presented with a set of questions. This set is relatively standard in the sense that answers to previous questions in the set do not normally influence whether a particular question is asked.

A clerical form is usually filled in by working from left to right and top to bottom. The person completing the form can make alterations whilst entering an answer, skip over questions temporarily, go back and change the answer to a previous question, or even tear up the form and start again. He maintains control up to the point when the form is handed over to the recipient.

Form dialogues typically provide the same facilities. A user may edit an individual answer as it is being entered. He may also move around the form, skipping questions or going back to answer a previous question. The user retains this ability until he indicates that he is satisfied with the input, either by pressing a particular key to accept the form or by answering a final question equivalent to

OK to process (y/n)

If the recipient of the form is present whilst the form is being completed, he can point out errors as they occur. This approach may distract the person completing the form and may lead to an increase in the number of errors; the alternative is to wait until the form has been completed before checking it. He will then indicate all the items which are wrong and ask for them to be corrected.

If the computer system is present, it can validate each answer immediately it is input, or it can wait and report errors only when the form has been completed. With some hardware configurations, the user's input is only available to the system when he presses an 'enter' key, typically at the end of the form. Whether to validate immediately or to defer is not a trivial decision;