Input / Output - Part 1

After our flight of fancy in the 'Just Do It' section, it's back to programming again.

We now come to a major part of any program - getting input from the user, and the output of results.

It sounds easy, but as we shall see, there are lots of problems for the programmer along the way.

Each program that you write processes 'data' and produces a result. So you need to have ways of getting data into the program.

Input can come from four main sources:

- User input typed into on-screen data areas.
- User input from various keys on the keyboard ('up arrow' for example)
- Input via a computer port, mouse, or joystick.
- Input from an existing 'File' of data records.

We will have a look at each of these in turn.

There are two variations - for a Console application – and for a Window program.

Console Input using Inkey$

The following is an example of an application that requires only one key to be pressed.

If you run the program, it displays a menu from which to select an option.

The **'inkey$'** function is within a loop, waiting for a key to be pressed. If you press any key other than one of the listed options 1, 2, 3, 4 or 9, nothing happens.

A validation check is made using the statement: **pos = instr("12349", a$)**

If you press a valid option key , (i.e. the character is somewhere in the string "12349"), the test loop exits and the **'Select'** group executes to display the operation which was chosen.

(The **'instr'** function returns zero if a character is not in the string, or returns it's position in the string if it is there).

Once the selection is made, the value 'pos' can be used to branch to a subroutine, or a section of the program which deals with that choice.

In this example, the program simply displays the user's choice. Notice that the value 'pos' is also used to get part of the confirmation message from the string array op[5]. It's always nice when you can use an input value for more than one purpose.

Nothing can go wrong with this type of user input. Even if he presses an invalid key, nothing will happen.

```
openconsole
def a$, op[5]:string
def pos:int
autodefine  "OFF"

Cls

op[1] = "Addition","Subtraction","Multiplication","Division"

locate 5,5: color 10,0
print "Select the calculation you want:" : print
color 7,0
print chr$(9) + "1  for Addition"
print chr$(9) + "2  for Subtraction"
print chr$(9) + "3  for Multiplication"
print chr$(9) + "4  for Division"
print
print chr$(9) + "9  to Exit"
print
print string$(40,"_")

do
   a$ = inkey$
' check whether the input character is in the list 1,2,3,4,9 ..
   if a$ <> "" then pos = instr("12349",a$)
until pos > 0

color 12,0: print

select a$
   case "1"
   case "2"
   case "3"
   case "4"
      Print "You selected: ", op[pos]
   case "9"
      Print
      Print "Press return to Exit"
endselect

do:until inkey$<>""
closeconsole
end
```

Console Input Using the Input Statement

This input method is used when you require the user to type several characters into the program.

You will normally offer a **'prompt'** to the screen, so that the user knows what he is expected to type.

You can either display the prompt first, followed by the input request (as in this example); or you can incorporate the prompt into the 'input' instruction.

```
openconsole
def name:string

autodefine "OFF"
cls
locate 5,5: color 14,0
print "Please enter your first name:" : print

do
locate 7,5: color 7,0
input name
until name <> ""

color 10,0: print
print space$(4) + "Hello ", name.

do:until inkey$<>""
closeconsole
end
```

A separate prompt allows better control of colour. With the single-line method, the prompt and the response have to be the same colour.

Console Colours

Colour Number	Colour
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Grey
9	Light Blue
10	Light Green
11	Light Cyan
12	Light Red
13	Light Magenta
14	Yellow
15	White

Window User Input

Now we will have a look at several methods of user-typed input into a Window program.

Users are awkward customers, so your program should expect just about anything to be typed in.

Some validation of input is nearly always necessary.

There are several Windows controls which will accept user-typed input.

The most obvious is an Edit Box, but there are several types - a Single -Line Edit Box, a Multi-line Edit Box, and a Rich Edit control.

User input to a Single-Line Edit Control

For our first example, we'll use a ‘Single-Line Edit Box’ into which we can type some input, and a ‘Text Box’ to display the result.

This should be the simplest approach to input a name, a password, or a numeric data value, but Microsoft have made things difficult.

The problem is, how does your program know when the user has finished typing the text or data, and wishes to enter the value into the program.

You would expect to type the information and then press the Enter(Return) key.

Unfortunately, Microsoft made no provision for the Enter key to send a message to Windows when it is pressed inside a single-line Edit Box control. If you try it, you'll just get an error beep.

That makes life rather difficult, and Creative Basic programmers have long sought workarounds for this problem.

There are several ways to proceed. Have a look at each of them and decide which best suits your application.

First, we will look at a program which responds to the ‘Return’ key being pressed.

What ! .. didn’t we just say that couldn’t be done?

Well it’s not straightforward. It will be necessary to invoke the Windows API *(Application Programming Interface)* library, and bring a ‘Timer’ into play. Big guns for such a simple task.

Not only that, but we shall not be able to get rid of the annoying error beep from the Edit box when the Return key is pressed.

However, in some circumstances, you may feel that this behaviour is acceptable ...

Using the Windows API function GetAsyncKeyState

The **GetAsyncKeyState** API function determines whether a specified key is up or down at the time the function is called, and whether the key was pressed after a previous call to **GetAsyncKeyState**.

If we use a timer to check the Return key every tenth of a second or so, we can tell when the Return key has been pressed.

```
def w:window
def wstyle,key,textW,textH:int
def a$:string

autodefine  "OFF"

declare "user32",Enter alias GetAsyncKeyState(vKey:int),int

wstyle = @SIZE|@MINBOX|@MAXBOX

window w,50,50,500,400,wstyle,0,"Single Line Edit Control",main
setwindowcolor w,rgb(0,0,50)

control w,"B, Exit,206, 300, 70, 35, 0, 1"

' Edit Box ...
control  w,"E,,190,50,110,25,@cteditcenter,2"
setcontrolcolor w,2,rgb(250,250,250),rgb(0,90,170)

' Text Box ...
control  w,"T,,188,150,110,25,@cteditcenter,3"
setcontrolcolor w,3,rgb(250,250,250),rgb(0,90,170)
setfont w,"Times New Roman",12,700,0,3

setfont  w, "Arial",10,500,0
a$ = "Enter your name and press Enter ..."
gettextsize w, a$, textW, textH
move w,(500 - textW)/2,20
frontpen w, rgb(20,230,250)
print w,a$

' a timer is needed to check for the Return keypress ...
starttimer w,100,1

setfocus w,2

waituntil w = 0

END
```

(Continued)

... continuation

```
SUB main
select @class
   case @IDClosewindow
' closing the window sets w = 0
' important to close any timers before closing the program ...
      stoptimer w,1
      closewindow w

' clicking the Exit button ...
   case @IDControl
      select @controlid
         case 1
            stoptimer w,1
            closewindow w
         case 2
' clear the Edit box whenever it gets the focus ...
            if @notifycode = @ensetfocus
               setcontroltext w,2,""
            endif
      endselect

   case @IDTimer
' check for the Return key (virtual key code:  hexadecimal 0D,  = 13) ...
      if Enter(13) <> 0
         a$ = getcontroltext(w,2)
         if a$ <> ""
            setcontroltext w,2,""
            setcontroltext w,3,a$
         endif
      endif
endselect
RETURN
```

Note the declaration of the Windows API function **GetAsyncKeyState( )** at the start of the program.

This line sets an Alias name **'Enter'** for the function (it's easier to type). Or you could use its full name **'GetAsyncKeyState '** to call the function if you prefer.

A **Timer** is set up to check for the 'Return' keypress, which will send a timer message every 100 milliseconds.

These messages are detected by the window's message handling routine, and filtered by the **case @idtimer** statement**.**

The **Enter( )** function is called, and this checks whether the **'Return'** key, (ASCII 13) has been pressed.

If it has, the text entered in the Edit box is read, and then written into the text box.

You will still get an 'error beep' when you press **'Return',** but the job is done.

Using a dedicated Enter Button

The next example demonstrates a more direct approach – using an 'Enter' button control to accept the input.

There is a robust feel to this method. The user types the data, and if necessary edits it.

None of the typing will be entered into the program until the user deliberately clicks the 'Enter' button. There is also the opportunity at this point to do validity checking, and to process any corrections, before accepting the data into the program.

One Enter button can be used to deal with a group of related Edit boxes if required.

```
def w:window
def a$:string
def wstyle:int

wstyle = @SIZE|@MINBOX|@MAXBOX
window w,50,50,500,400,wstyle,0,"Simple Edit Box with Enter Button",main
centerwindow w

control w,"E,,188,50,110,25,@cteditcenter,1"
control w,"B,Exit,206,300,70,35,0,2"
control w,"T,,185,200,110,22,@cteditcenter,3"
control w,"B,Enter,206,130,70,35,@CTLBTNFLAT,4"
setcontrolcolor w,1,rgb(250,250,250),rgb(0,90,170)
setcontrolcolor w,3,rgb(230,230,100),rgb(0,90,170)
setcontrolcolor w,4,rgb(0,200,0),rgb(0,40,100)
setwindowcolor w,rgb(0,0,20)

setfocus w,1

waituntil w = 0
END

SUB main
select @class
   case @IDClosewindow
      closewindow w
   case @IDcontrol
      select @controlid
         case 1
            if @notifycode = @ensetfocus
' clear the edit box when it gets the focus ...
               setcontroltext w,1,""
               setcontroltext w,3,""
            endif
         case 2
            closewindow w
         case 4:' Enter button pressed ...
            a$ = ltrim$(getcontroltext(w,1))
            if a$ <> ""
               setcontroltext w,1,""
               setcontroltext w,3,a$
               setfocus w
            else
               setfocus w,1
            endif
      endselect

endselect
RETURN
```

The Enter button method, requires a dedicated button, and a short 'Button Press' section of code to process the data from the Edit box.

The Edit box is a simple single-line box with no special arrangements.

It will not respond to a 'Return' key press – it just beeps a warning message.

This method is the simplest, and will probably be the one you will use most often.

So is that it for entering data into Edit Boxes?

Oh .. no .. there are some other strange ways, discovered by enterprising programmers over the years. Who knows, you might like to use them yourself.

Using a Multi-Line Edit Control

Here's another method, this time using a Multi-Line Edit control instead of the simple single line Edit Box.

It's still an Edit Box, so we won't be able to detect whether the **Enter** key has been pressed.

So how does that help? Well, pressing the Enter key when the **@cteditmulti** flag is used will move the cursor to the next line, and we shall be able to detect that. When a change from one, to two lines occurs, that will be the signal that the Enter key has been pressed.

Here's the code:

```
def w:window
def wstyle,estyle,key,count:int
def textW,textH,max2 as int
def a$:string

autodefine = "OFF"

wstyle = @SIZE|@MINBOX|@MAXBOX
a$ = "Enter Key pressed in MultiLine Edit Control"

window w,50,50,500,400,wstyle,0,a$,main
setwindowcolor w,rgb(0,0,50)

control w,"B, Exit,206, 300, 70, 35, 0, 1"

' Edit Box ...
estyle = @cteditcenter|@cteditmulti|@tabstop|@cteditautov
control w,"E,,190,50,110,25,estyle,2"
setcontrolcolor w,2,rgb(250,250,250),rgb(0,90,170)
```

(Continued)

... continuation

```
max2 = 5   :' set max number of characters

' allow for additional CR + LF characters ..
controlcmd w,2,@EDSETLIMITTEXT,max2 + 2

' Text Box ...
control w,"T,,188,150,110,25,@cteditcenter,3"
setcontrolcolor w,3,rgb(250,250,250),rgb(0,90,170)
setfont w,"Times New Roman",12,700,0,3

setfont w, "Arial",10,500,0
a$ = "Enter your first name and press Enter ..."
gettextsize w, a$, textW, textH
move w,(500 - textW)/2,20
frontpen w, rgb(20,230,250)
print w,a$

setfocus w,2

WAITUNTIL w = 0
END

SUB main
select @class
   case @IDClosewindow
' closing the window sets w = 0
      closewindow w
' clicking the Exit button ...
   case @IDControl
      if @controlid = 1 then closewindow w
      if @controlid = 2
' deal with an Edit box event ..
' trim any CR + LF characters
         a$ = rtrim$(getcontroltext(w,2))
' how many lines are there ? ..
         count = controlcmd(w,2,@EDGETLINECOUNT)
         if count = 2        :' Enter key was pressed ..
            if a$ <> ""      :' do nothing if the edit box is blank
               setcontroltext w,3,a$
            endif
            setcontroltext w,2,"" :' reset the edit box
         else
' delete any characters typed beyond the maximum length
            controlcmd w, 2, @EDSETSELECTION, max2,max2+1
            controlcmd w, 2, @EDDELETESEL
         endif
      endif
endselect
RETURN
```

We are not really interested in the second line of the Edit Box - it's just an indicator.

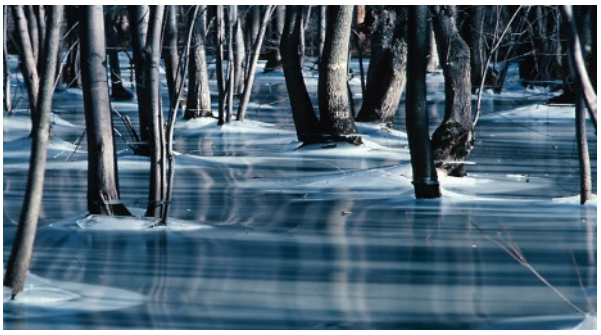
When the Enter key is pressed, two characters are placed at the end of the first line. These are the **Carriage Return** chr$(13), and **Line Feed** chr$(10) characters.

The cursor then moves to the second line, and we can now detect two lines, which means the **Enter** key has been pressed.

The code ignores an empty Edit box, and prevents the number of characters entered from exceeding the specified maximum (in this example, 5 characters).

The contents of the Edit box are transferred to the Text box when the **Enter** key is pressed.

Validation and any possible error messages would be dealt with at this point.



Using a Rich Edit Control

Here's another method this time using a Rich Edit control instead of a simple Edit Box.

It looks like an ordinary Edit box to the user, but the Rich Edit control has many more features. (You can read about the Rich Edit control details in the User's Guide).

It requires a bit of tricky programming to make it function as an Edit box, and be able to respond to the user pressing the **Enter**(Return) key, .

We shall need to detect keyboard events, apply a User Data Type, and use some memory operations.

Although it's a bit less straightforward than the previous methods, it works well.

Here's the code:

```
def w:window

' Setting for rich edit control to process key events ...
setid "ENMSGFILTER",0x700

type MSGFILTER
     def hwndFrom:int
     def idFrom:int
     def code:int
     def msg:int
     def wparam:int
     def lparam:int
endtype

def mf:MSGFILTER  :' mf is now a user defined variable of type MSGFILTER
def mem:memory

def left,top,width,height,textW,textH,run,key:int
```

(Continued)

... continuation

```
def a$:string

window w,0,0,600,400,0,0,"Use of Rich Edit Enter Key", mainwin
setwindowcolor w,rgb(0,20,50)

getclientsize w,left,top,width,height

' Rich Edit Box ...
control w,"RE,,(width-80)/2,90,80,25,@cteditcenter,1"
setcontrolcolor w,1,rgb(0,0,0),rgb(0,150,200)
controlcmd w,1,@RTSETEVENTMASK,@ENMKEYEVENTS
controlcmd w,1,@RTSETDEFAULTFONT,"Times New Roman",14,1,0

' Text Box ...
control w,"T,,(width-80)/2,150,80,25,@cteditcenter,2"
setcontrolcolor w,2,rgb(0,0,0),rgb(0,150,200)
setfont w,"Times New Roman",14,700,0,2

control w,"B,Exit,(width-50)/2,300,50,25,0,3"

setfocus w,1

setfont w, "Arial",10,500,0
a$ = "Enter some Text and press Enter ..."
gettextsize w, a$, textW, textH
move w,(width- textW)/2,40
frontpen w, rgb(20,230,150)
print w,a$

run = 1

waituntil run = 0
closewindow w

END

SUB mainwin
   select @class
      case @IDClosewindow
         run = 0
      case @IDControl
         select @controlid
            case 1
               if @notifycode = @ENMSGFILTER

' read in the MSGFILTER structure ..
                  mem = @qual
                  readmem mem,1,mf

' at this point the keyboard event
' is in mf.msg and the key code is in mf.wparam
' the event can be things like @IDCHAR
```

(Continued)

... continuation

```
      if mf.msg = @idchar
                    key = mf.wparam
                    if key = 13
                       a$ = getcontroltext(w,1)
                       if ltrim$(a$) <> ""
                        setcontroltext w,1,""
                        setcontroltext w,2,a$
                       else
                         setcontroltext w,1,""
                       endif
                    endif
                 endif
              endif
              if @notifycode = @ensetfocus
                setcontroltext w,1,""
              endif
           case 3
             run = 0
        endselect
   endselect

RETURN
```

The event mask **@RTSETEVENTMASK** specifies which notification messages the **RichEditCtrl** object sends to its parent window.

That's quite a long piece of code, but if you copy and paste the three sections one after the other into a new source file, you should find that having entered some text in the Rich Edit box, pressing the **Enter** (Return) key will copy it into the text box.

So far, we've seen five different ways of getting user-typed data into your Windows program:

- Using a Single-Line Edit Box
- Using the API function GetAsyncKeyState
- Using a dedicated Enter Button
- Using a Multi-Line Edit Box
- Using a Rich Edit Control

Are there any other methods?

Of course there are, and they get more wild and wonderful as we go on.

So for the sake of completeness, onwards we go ..

User input to a Single Line Edit Control - Mark II

Just a minute .. wasn't this the first method we looked at?

Yes it was - and if you remember, we had to use an API function **GetAsyncKeyState**, and a timer to make it work.

The method we'll look at now is very strange. How anyone ever discovered it I can't imagine (cheers Lefty!)

Although an Edit Control cannot report that the Enter key has been pressed to it's parent window, another kind of message can be detected.

The amazing thing is that this is done by checking for a **Menu** operation …
case @IDMENUPICK - whether or not there actually is a menu.

A response of **@IDOK** corresponds to the **Enter** key being pressed, and **@IDCancel** corresponds to the **ESC**(ape) key being pressed.

That's a pretty surprising finding. Here's the code using a two Edit box example, which also allows the Tab key to be used.

```
' Enter Button trapped for edit boxes ..

def w:window
def run,wstyle,Focus:int
def a$:string

autodefine = "OFF"

wstyle = @SIZE|@MINBOX|@MAXBOX

window w,0,0,310,200,wstyle,0,"Enter Key Trapping",messages
setwindowcolor w,rgb(0,0,70)
enabletabs w,1

' the method seems to work with or without the menu ..
' menu w,"T,&File,0,0","I,&Exit,0,1","I,&Exit2,0,2"

control w,"E,,60,20,80,24,@tabstop|@cteditcenter,1"
control w,"E,,170,20,80,24,@tabstop|@cteditcenter,2"
control w,"T,,113,80,80,24,@cteditcenter|0x200,3"
control w,"B,Exit,(310-70)/2,120,70,30,0,4"

setfocus w,1

run = 1
waituntil run = 0
closewindow w
end
```

(Continued)

... continuation

```
sub messages
select @class
   case @IDMENUPICK
' deal with normal menu options ..
      if @menunum = 1 then run = 0
      if @menunum = 2 then run = 0
' check for Enter key pressed ..
      if @code = @IDOK
         select Focus
            case 1
               a$ = getcontroltext(w,1)
               setcontroltext w,3,a$
               setfocus w,2
            case 2
               a$ = getcontroltext(w,2)
               setcontroltext w,3,a$
               setfocus w
         endselect
      endif
' check for ESC key to exit ..
      if @code = @IDCancel then run = 0
   case @IDclosewindow
      run = 0
   case @IDcreate
      centerwindow w
   case @IDcontrol
      if @controlid = 4 then run = 0
      if @notifycode = @ENSETFOCUS
         select @controlid
            case 1
               setcontroltext w,1,""
               Focus = 1
            case 2
               setcontroltext w,2,""
               Focus = 2
         endselect
      endif
endselect
return
```

The method works because Creative Basic allows a menu item to be executed by pressing the Enter key - even if no item is selected.

It's a bit devious, but it seems to work fine.

Using Ghost Text Boxes

If you thought the last method was weird, wait until you see this one, which has **no Edit Box at all**.

Here we have decorative editable areas, and a single 'ghost' Text Box, which moves around the screen as required.

This example uses three text areas, but there could be lots of them forming a table.

There would still only be one actual Text Box required.

```
' Ghost Edit using a single Text box ...
' Edit placeholders are just decorative locations.
' They can be any decorative image (like a Web button)
' The Text box moves to these locations as required ..

def w:window
def left,top,width,height,textW,textH,chartest:int
def i,j,tboxW,tboxH,bc,test1,test2,run:int
def highbox,tb,tb2,mx,my,key:int
def a$,b$,ok1$,ok2$:string

type ghosttype
   def l:int
   def t:int
   def w:int
   def h:int
   def text:string
   def nxt:int
   def max:int
   def ok:string
endtype

def tbox[3]:ghosttype

window w,0,0,600,400,@SIZE|@MINBOX|@MAXBOX,0,"Text Box Edit Example",main
centerwindow w
setwindowcolor w,rgb(0,0,30)
setfocus w

getclientsize w,left,top,width,height

' set the size of the editable text box and create it ...
tboxW = 81: tboxH = 21
control w,"T,,120,70,tboxW,tboxH,@cteditcenter,1"
setcontrolcolor w,1,rgb(230,230,230),rgb(0,100,200)
setfont w,"Times New Roman",10,600,0,1
' hide the text box ...
SHOWWINDOW w, @swhide,1

control w,"B,Exit,(width-70)/2,300,70,35,0,2"
control w,"T,,(width-80)/2,180,80,22,@cteditcenter,3"
setcontrolcolor w,3,rgb(240,240,100),rgb(0,100,200)
setfont w,"Times New Roman",10,600,0,3
```

(Continued)

... continuation

```
' load your own suitably sized jpg image for a nice placeholder background.
' im = LOADIMAGE (GETSTARTPATH + "box.jpg", 4)
' this has been replaced in this example by a line drawing section below to
' give a suitable effect ...

' locate the editable text boxes and initialise ...
tbox[0].l = 100: tbox[1].l = 255: tbox[2].l = 410
tbox[0].t = 70: tbox[1].t = 70: tbox[2].t = 70
tbox[0].w = 90: tbox[1].w = 90: tbox[2].w = 90
tbox[0].h = 25: tbox[1].h = 25: tbox[2].h = 25
' set next box when Enter is pressed ...
tbox[0].nxt = 1: tbox[1].nxt = 2: tbox[2].nxt = 99
' set maximum number of characters for each box ...
tbox[0].max = 5: tbox[1].max = 5: tbox[2].max = 5
ok1$ = "1234567890+-"            :' set up valid characters for each box ...
ok2$ = "aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ,."
tbox[0].ok = ok1$                :' set up left hand box for numbers only
tbox[1].ok = ok2$                :' set up right hand box for letters only ...
tbox[2].ok = ""                  :' set up third box for any character type ...
highbox = 2                      :' set highest index ...

' show the ghost placeholders ...
for i = 0 to highbox
' ____________________________________________________________________________
' this line drawing section would normally be replaced by an image background
' loaded from a suitable image, sized to fit.
   for j = 0 to tbox[i].w
      if j < tbox[i].w /2
         bc = 100+(200*j/tbox[i].w)
      else
         bc = 300-(200*j/tbox[i].w)
      endif
      line w,tbox[i].l+j,tbox[i].t,tbox[i].l+j,tbox[i].t+tbox[i].h,rgb(0,0,bc)
   next j
' ____________________________________________________________________________
' the image would normally replace the above line drawing section ...
'  ShowImage w,im,4,tbox[i].l,tbox[i].t,tbox[i].w,tbox[i].h

next i

' invalidate textbox indexes, so as to start with no box selected ...
tb = -1: tb2 = -1

' ****** documentation only ******************
setfont w, "Arial",10,500,0
frontpen w, rgb(20,230,150)
a$ = "Select box - enter some data and press Enter - clears if clicked"
gettextsize w, a$, textW, textH
move w,(width- textW)/2,20
print w,a$

setfont w, "Arial",10,500,0
a$= "(Box 1, 5 digits max -  Box 2, 5 letters max -  Box 3, any 5 characters)"
gettextsize w, a$, textW, textH
move w,(width- textW)/2,120
print w,a$
' ****** documentation only ******************
```

(Continued)

... continuation

```
frontpen w, RGB(50,100,250)

run = 1

WAITUNTIL run = 0
' image delete if a jpeg background image was loaded ...
' DeleteImage im,4
closewindow w
END

SUB main
select @CLASS
   case @IDCHAR
      key = @CODE
      select 1
         case (key = 9)
' ignore TAB ...
            return
         case (key <> 13) & (key <> 8)
' normal character entered ..
            if (tb >= 0) & (tb <= highbox)
               a$ = getcontroltext(w,1)
               b$ = chr$(key)
               if (tbox[tb].ok <> "")
                  chartest = (instr(tbox[tb].ok,b$) > 0)
               else
                  chartest = 1
               endif
               if (len(a$) < tbox[tb].max) & chartest
                  a$ = a$ + chr$(key)
               setcontroltext w,1,a$
               endif
            endif
         case (key = 8)
' backspace pressed ...
            a$ = getcontroltext(w,1)
            sl = len(a$)
            if sl > 0
               a$ = left$(a$,len(a$)-1)
            setcontroltext w,1,a$
            endif
         case (key = 13)
' Enter key pressed ..
            if (tb >= 0) & (tb <= highbox)
               writetext
               if len(a$) > 0
                  setcontroltext w,3,a$
                  tb = tbox[tb].nxt
                  textmove
               endif
            endif
      endselect
   case @IDCLOSEWINDOW
      run = 0
```

(Continued)

... continuation

```
   case @IDCONTROL
      select @CONTROLID
         case 2
            run = 0
      endselect
   case @IDLBUTTONDN
' if mouse clicked, find where ...
      tbox_click
   case @IDMOUSEMOVE
    mx = @mousex : my = @mousey
ENDSELECT
RETURN

SUB tbox_click
' check for mouse click in a box ...
def tbv:int
 for tbv = 0 to highbox
   test1 = (mx>tbox[tbv].l) & (mx<(tbox[tbv].l+tbox[tbv].w))
   test2 = (my>tbox[tbv].t) & (my<(tbox[tbv].t+tbox[tbv].h))
   if test1 & test2
' transfer the contents of the text box ...
         if tb <= highbox
            writetext
         endif
         setcontroltext w,3,""
         tb = tbv
         textmove
   endif
 next tbv
RETURN

SUB textmove
' move the text box to the ghost placeholder clicked ...
' remove focus from current box ..
   if tb >= 0
      if tb <= highbox
         setcontroltext w,1,""
' repaint the ghost background ...
' an image would normally be used ...
' ---------------------------------------------------------------
         for j = 0 to tbox[tb].w
            if j < tbox[tb].w /2
               bc = 100+(200*j/tbox[tb].w)
            else
               bc = 300-(200*j/tbox[tb].w)
            endif
            line w,tbox[tb].l+j,tbox[tb].t,tbox[tb].l+j,tbox[tb].t+tbox[tb].h,rgb(0,0,bc)
         next j
' ---------------------------------------------------------------
'         ShowImage w,im,4,tbox[tb].l,tbox[tb].t,tbox[tb].w,tbox[tb].h

' display the text box at the clicked location ...
         showwindow w, @swrestore,1
         setsize w,tbox[tb].l+5,tbox[tb].t+2,tboxW,tboxH,1
         tb2 = tb
      else
' if last box in the sequence just go to the window ...
' and hide the text box ...
         showwindow w, @swhide,1
         setfocus w
         tb2 = -1
      endif
   endif
RETURN
```

... continuation

```
SUB writetext
' transfer text from the text box to the selected ghost text area ...
   a$ = getcontroltext(w,1)
   if len(a$) > 0
      setfont w,"Times New Roman",10,600
      frontpen w, rgb(0,200,255)
      gettextsize w, a$, textW, textH
' note the divide by 3 to centre the text vertically ...
      move w,tbox[tb].l+(tbox[tb].w-textW)/2,tbox[tb].t+(tbox[tb].h-textH)/3
      drawmode w, @transparent
' write the information to the screen...
      print w,a$
' and save it in the ghost text area ...
      tbox[tb].text = a$
' clear the text box ...
      setcontroltext w,1,""
   endif
RETURN
```

If we can have ghostly text areas (they are not real controls - they could be just images like scrolls of parchment for example), as programmers we can make them do anything.

Suppose in a lottery, there are six lottery balls for a given week. The user types in the date, and six numbers, which now display in seven virtual boxes on screen.

When he presses the Enter key, the program could fade out the seven ghost boxes, re-arrange the data into one comma separated string (for storage), and display it in one new elongated ghost box ....

Or, if there was a wrong value in the set, the ghost text area could change shape, and display a rude flashing ‘try again' message ...

Or, to impress the user, the individual ghost boxes could be moved to another place on the screen - or pop up again on another screen sometime later ....

Or, ... Well, I probably missed lots of other things you could do with ‘ghost’ images ...

Is that fun or what? Instead of boring Edit Boxes, you can have nice decorative areas that the user can ‘apparently’ type into.

The downside is that you have to program the editing facilities yourself.

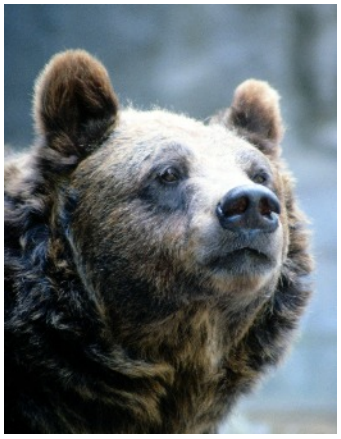
Well, that's about it for user data entry into windows.

Direct User Input

In this section we will look at input coming from the keyboard, the mouse, or a joystick.

This type of input will usually have some immediate effect on your program – real time interaction.

Input from various keys on the keyboard



We have seen previously that Windows works by using a messaging system. So you will not be surprised to learn that when you press a key on the keyboard, messages are sent to the Window's message queue.

Messages from the keyboard are generated whenever a key is pressed while your window has focus. Keyboard messages are sent in two different forms, keystroke messages and character messages.

Each time a key is pressed, a message is sent to the active window (**@IDKeydown**), and when the key is released, another message is sent (**@IDKeyup**).

Creative can detect the ASCII value of any of the keys which generate a printable character, using the **@idchar** message.

This contains the system variables **@code** and **@qual.**. (**@code** contains the ASCII value).

Here's an example fragment to detect the **Esc(ape)** key (Ascii 27):

```
SUB messages
select @class
   case @idchar :' pressing the Esc(ape)key will abort the program
      key = @code
      if key = 27 then run = 0
   case @idclosewindow
      run = 0
endselect
RETURN
```

Keys such as arrow keys, function and keypad keys, do **not** provide an ASCII value.

Many such keys are accessible using "**virtual key codes**". (see the User Guide - Appendix for a list of these codes).

For example, the '**Shift**' key has been assigned the (Hex) code **0x10** (or 16 in decimal).

If you need to access the non-ascii keys, you have to use the **@idkeydown** and **@idkeyup** messages provided by Creative Basic.

In Windows programs, you can also use the **GETKEYSTATE()** function to detect whether a particular key on the keyboard is being pressed when the function is executed. (Note: This will not work in a Console-only program).

Here's a program to show which key is pressed on the keyboard, using **@IDChar** for printable keys, and **@IDKeydown** for non-printable characters ..

```
' Keyboard Input Version 1 ..
' GWS 2010

def w:window
def run, textW, textH:int
def a$,vk[256]:string

autodefine "OFF"

' open the window ..
window w, -700, 0, 700, 450, @SYSMENU, 0, "Keyboard Input", msghandler
setwindowcolor w, rgb(0,100,120)

' define a Button control ..
control w, "B,Exit, (700 - 70)/2, 320, 70, 30, @ctlbtnflat, 1"
setcontrolcolor w, 1, 0, rgb(120,140,220)

' display some text ..
a$ = "Press any Key"
setfont w, "Arial", 30, 700, @sfitalic
frontpen w, rgb(130,130,230)
drawmode w, @transparent
gettextsize w, a$, textW, textH
move w,(700-textW)/2,60
print w, a$

' set up the virtual key descriptions ..
vk[12] = "Clear"
vk[16] = "Shift","Ctrl","Alt","Pause","Caps Lock"
vk[33] = "Page Up","Page Down","End","Home"
vk[44] = "PrtScr","Insert","Delete"
vk[37] = "Left Arrow","Up Arrow","Right Arrow","Down Arrow"
vk[112] = "F1","F2","F3","F4","F5","F6","F7","F8","F9","F10","F11","F12"
vk[144] = "Num Lock","Scroll Lock"

run = 1

waituntil run = 0
closewindow w
end

sub msghandler
select @class
   case @idclosewindow
      run = 0
   case @idcreate
      centerwindow w
   case @idcontrol
      if (@controlid = 1) then run = 0
   case @IDChar
' to detect keys which generate printable values ..
      setfont w, "Arial", 20, 700
      frontpen w, rgb(200,200,30)
      drawmode w, @transparent
' key description ..
      rect w,100,150,500,35,rgb(0,100,120),rgb(0,100,120)
```

(Continued)

```
      select @code
         case 8
            a$ = "Key pressed is: Backspace"
         case 9
            a$ = "Key pressed is: Tab"
         case 13
            a$ = "Key pressed is: Return"
         case 27
            a$ = "Key pressed is: Esc"
         case 32
            a$ = "Key pressed is: Space"
         default
            a$ = "Key pressed is: " + chr$(@code)
      endselect

      gettextsize w, a$, textW, textH
      move w,(700-textW)/2,150
      print w, a$
' key ascii value ..
      rect w,100,200,500,35,rgb(0,100,120),rgb(0,100,120)
      a$ = "ASCII code is: " + str$(@code)
      gettextsize w, a$, textW, textH
      move w,(700-textW)/2,200
      print w, a$

   case @IDKeydown
      setfont w, "Arial", 20, 700
      frontpen w, rgb(200,200,30)
      drawmode w, @transparent
' key description for virtual keys ..
      rect w,100,150,500,35,rgb(0,100,120),rgb(0,100,120)
      a$ = "Key Pressed is: " + vk[@code]
      gettextsize w, a$, textW, textH
      move w,(700-textW)/2,150
      print w, a$
' key hex value ..
      rect w,100,200,500,35,rgb(0,100,120),rgb(0,100,120)
      a$ = "Virtual Keycode is: " + hex$(@code) + "(hex)"
      gettextsize w, a$, textW, textH
      move w,(700-textW)/2,200
      print w, a$

endselect
return
```

You may notice that the “PrtScr” key does nothing. I’ve not found an answer to that. It doesn’t seem to react with the program in any way.

The keyboard utility program works fine - but it's an example of what you might call a 'direct' way of programming.

You think what you want to achieve, and just go for it, using the most obvious tools available.

In this application, I wanted two lines of text to appear each time a key is pressed. The first line to show the character, or description of the key.

The second line to show either the Ascii value, or the Virtual Key Code, whichever is appropriate.

The obvious tools in this case are to set up the text string, set the Font, set the Colour, set the screen location, and finally to write the text to the screen.

Of course, if the user presses another key, the same thing happens, and the second lines of text will overwrite the original ones - leaving a nasty jumble of characters.

That's the reason for having the Rectangle statement, which blanks the text area of the screen to give a clean slate each time.

Now we have a working version of the program, it comes to mind that there might be a better way to do it. Maybe we could save some statements, maybe re-arrange the code in some way, maybe use some subroutines to tidy it up .. Many possibilities.

Suppose we use two **Text boxes** to hold the text - that's a fairly obvious step.

Text boxes are easy to set up, and you only have to set the Font Size and Colour once.

Now we can dispense with the blanking rectangles, since new text will automatically overwrite any previous text in the boxes. Text can also be specified as centred - and that only needs to be done once.

Was it worth writing the 'direct' version? Yes, because the logic was developed and tested.

All that is needed now is to 're-arrange the furniture' a bit - loading the text into the text boxes, and deleting quite a few statements which are now superfluous.

So let's make those changes, and see how the program looks ..

Keyboard Input - Version 2

```
' Keyboard Input Version 2 ..
' GWS 2010

def w:window
def run, textW, textH:int
def a$,vk[256]:string

autodefine "OFF"

' open the window ..
window w, -700, 0, 700, 450, @SYSMENU, 0, "Keyboard Input", msghandler
setwindowcolor w, rgb(0,100,120)

' define a Button control ..
control w, "B,Exit, (700 - 70)/2, 320, 70, 30, @ctlbtnflat, 1"
setcontrolcolor w, 1, 0, rgb(120,140,220)

'define two text boxes ..
control w, "T,, (700 - 400)/2, 150, 400, 50, @cteditcenter|0x200, 2"
control w, "T,, (700 - 400)/2, 210, 400, 50, @cteditcenter|0x200, 3"

for i = 2 to 3
   setcontrolcolor w,i,rgb(200,200,30),rgb(0,100,120)
   setfont w, "Arial", 20, 700, 0, i
next i

' display heading text ..
a$ = "Press any Key"
setfont w, "Arial", 30, 700, @sfitalic
frontpen w, rgb(130,130,230)
drawmode w, @transparent
gettextsize w, a$, textW, textH
move w,(700-textW)/2,60
print w, a$

' set up the virtual key descriptions ..
vk[12] = "Clear"
vk[16] = "Shift","Ctrl","Alt","Pause","Caps Lock"
vk[33] = "Page Up","Page Down","End","Home"
vk[44] = "PrtScr","Insert","Delete"
vk[37] = "Left Arrow","Up Arrow","Right Arrow","Down Arrow"
vk[112] = "F1","F2","F3","F4","F5","F6","F7","F8","F9","F10","F11","F12"
vk[144] = "Num Lock","Scroll Lock"

run = 1

waituntil run = 0
closewindow w
end
```

(Continued)

... continuation

```
sub msghandler
select @class
   case @idclosewindow
      run = 0
   case @idcreate
      centerwindow w
   case @idcontrol
      if (@controlid = 1) then run = 0
   case @IDChar
' to detect keys which generate printable values ..
' key description ..
      rect w,100,150,500,35,rgb(0,100,120),rgb(0,100,120)
      select @code
         case 8
            a$ = "Key pressed is: Backspace"
         case 9
            a$ = "Key pressed is: Tab"
         case 13
            a$ = "Key pressed is: Return"
         case 27
            a$ = "Key pressed is: Esc"
         case 32
            a$ = "Key pressed is: Space"
         default
            a$ = "Key pressed is: " + chr$(@code)
      endselect
      setcontroltext w,2,a$
' key ascii value ..
      a$ = "ASCII code is: " + str$(@code)
      setcontroltext w,3,a$

   case @IDKeydown
' to detect keys which generate non-printable values ..
' key description for virtual keys ..
      a$ = "Key Pressed is: " + vk[@code]
      setcontroltext w,2,a$
' key hex value ..
      a$ = "Virtual Keycode is: " + hex$(@code) + "(hex)"
      setcontroltext w,3,a$
endselect
return
```

I think you will agree that this is a much neater method.

Did you notice that the two text boxes were formatted using a ‘FOR’ loop ?

There were only two controls in this example, but the loop method is useful when there are several controls, all sharing the same specifications.

Direct user input

Now we'll take a look at input coming from the keyboard, the mouse, or a joystick in a game environment.

User input of this type will usually have an immediate effect on your program, resulting in real-time interaction.

Games, sprites and animation use Window's DirectX to communicate directly with the video hardware. This ensures smooth animation, fast screen updating, and rapid response to user input..

Keyboard Input

The keyboard provides us with a set of buttons, each of which can be used to control a game.

There are two ways to detect keyboard activity - either wait for a Windows message, or continually interrogate a key to see if it is currently being pressed.

When the user presses a key, an **@IDKEYDOWN** message is sent, with the system variable **@code** holding the corresponding hexadecimal key code.

(see the Creative user guide Appendix C for a list of all the key codes)

Using this method, your program is effectively waiting for a 'keyboard event interrupt'.

Here is a list of some of the keys often used to control games ..

Some Key Codes

Key	Hex Code
Up arrow	0x26
Down arrow	0x28
Left arrow	0x25
Right arrow	0x27
Control (Ctrl)	0x11
Shift	0x10
Spacebar	0x20
Enter	0x0D
Escape (Esc)	0x1B

This code snippit shows how such a keyboard message can be processed ..

```
messages:

select @CLASS
  case @IDCLOSEWINDOW
      run = 0
  case @IDKEYDOWN
      if (@code = 0x1B) then run = 0      :' Escape key ends game
    ..........

endselect
RETURN
```

The ‘messages’ approach does work, but the response you get is not very smooth.

Windows has a lot of messages to process, and the one you’re interested in may be slightly delayed.

When you need rapid responses to control a game, the second method of interrogating the state of a key using DirectX is much better.

The command **GetKeyState(keycode)** will do the job.

Keyboard Input using DirectX

If your application requires moving graphics, sprites or 3D, you will need to use DirectX to access the video card ram and its high speed features.

The **GetKeyState(keycode)** command only returns the state of a key at the time the command is executed So it needs to be repeatedly performed, looking for the key to be pressed.

This seems to be a problem, maybe requiring a timer - but there is a better method.

When you begin a DirectX program, you first open a normal Window - for example:

```
style = @NOAUTODRAW|@MINBOX|@MAXBOX|@SIZE
window win,0,0,width,height,style,0,"caption",messages
```

Then you need to open a DirectX screen using the CREATESCREEN command.

```
' create a DX screen ..
CREATESCREEN(win,width,height)
```

When your program runs, you will get **@IDDXUPDATE** messages sent to your message handling routine at intervals **of a** few milliseconds. These messages are used to implement a ‘game loop’, which provides the rapid graphics updates to the screen .

The game loop is the ideal place for the **GetKeyState(keycode)** command, and this means the state of a specified key will be examined frequently.

```
' create a DX screen ..
CREATESCREEN(win,width,height)
```

This code snippit shows how such a keyboard message can be processed ..

```
messages:

select @CLASS
  CASE @IDDXUPDATE
      if getkeystate(0x26)     :' up arrow pressed - so move Up

      ................
      endif

endselect
RETURN
```

So now we need an example of how this comes together.

But before we get started, I have to explain a change in presentation for this and subsequent guides.

So far, all discussions have been possible with some text, followed by an example program inserted in the text.

You may have noticed that the example programs have been getting longer as we progressed.

Now we are becoming involved with graphical methods, the code examples will require graphics and sound resources (images and music). I can't include such things in a simple text document.

So I think the best approach is to separate out the code and required resources for each example project into named .zip files which I can include with the .zip of this document.

In the document text, I'll use snippets to discuss any significant sections of the source code.

The full program and it's resources will be included in the associated .zip file.

I hope this will work out OK.

Now we'll look at a little game that illustrates how key presses can be used for control.

The Balloons Game

This is a game where the objective is to keep the balloon out of harm's way from the pins.

Like most games involving moving sprites, it uses a Direct X screen in a window.

As we saw earlier, this provides an ideal way of detecting user key presses, using the update messages for the DX screen together with the **getkeystate( )** command**.**

So let us examine the control section of the code for this game.

Here's a snippit of the code ..

```
sub update

' check for balloon reaching screen edges ..
   if getkeystate(0x27)                        :' right arrow pressed - move right
      if (bx < (sw - 100)) then bx = bx + bspeed * dt
   endif

   if getkeystate(0x25)                        :' left arrow pressed - move left
      if (bx > 0) then bx = bx - bspeed * dt
   endif

   if getkeystate(0x26)                        :' up arrow pressed - move Up
      if (by > 5) then by = by - bspeed * dt
   endif

   if getkeystate(0x28)                        :' down arrow pressed - move down
      if (by < (sh - 105)) then by = by + bspeed * dt
   endif
```

You can see that the **getkeystate( )** functions are used to detect when the user presses any of the arrow keys.

In fact, you can press an Up arrow key and a Right arrow key at the same time, to get a diagonal movement.

The other method of detecting key presses that we mentioned earlier was by detecting a **@IDKEYDOWN** message.

This technique is used in the game to detect if the user presses the ESCape key. Here's the snippet ..

```
Mainwindow:

select @CLASS
   case @IDCLOSEWINDOW
      run = 0

   case @IDKEYDOWN
      if (@code = 0x1B) then run = 0       :' Escape key ends game
```

The full working version of the Balloons program is in the Balloons.zip file attached to this document. We will return to this program later, when we look at graphics programming in Creative Basic.

So that covers Keyboard input into Windows programs - games in particular.

Now we need to look at input using the mouse.

Mouse Input

To detect user mouse activity, Creative offers a number of mouse messages which can be used to take appropriate actions.

The mouse generates input events whenever the user moves the mouse, or presses or releases a mouse button.

When a mouse message is received the system variables **@MOUSEX** and **@MOUSEY** will contain the position of the cursors hot spot at the time the message was generated.

You will only receive mouse messages when the cursor is within the window's client area.

Here are the mouse messages that can be used ..

Mouse Messages

Message ID	Meaning	Windows equivalent
@IDMOUSEMOVE	The mouse was moved.	WM_MOUSEMOVE
@IDLBUTTONDN	Left mouse button was pressed.	WM_LBUTTONDOWN
@IDLBUTTONUP	Left mouse button was released.	WM_LBUTTONUP
@IDLBUTTONDBLCLK	Left mouse button was double clicked.	WM_LBUTTONDBLCLK
@IDRBUTTONDN	Right mouse button was pressed.	WM_RBUTTONDOWN
@IDRBUTTONUP	Right mouse button was released.	WM_RBUTTONUP
@IDRBUTTONDBCLK	Right mouse button was double clicked.	WM_RBUTTONDBLCLK

The following test program will illustrate how these messages can be used. You need to load the program from the attached **mouse.zip** file to see the button sprite.

It also demonstrates how, in a Direct X screen, a user's own image of a button can be used to exit the program.

Why bother you might ask.? Well, it appears that Microsoft does not advocate using standard GDI controls with a Direct X screen. Which is a pity since it's very easy to do.

(In fact it appears to work with no problems in both Creative and IWBasic).

The example program uses standard text boxes to display the mouse co-ordinates as it moves around the screen, and a .jpg image sprite of a button.

When the cursor is over the button image, you can click the mouse, and the program will exit.

It does this by tracking the cursor with a small 2x2 pixel point sprite that is not visible, and when the left mouse button is clicked, the program checks for a collision between the point sprite and the button sprite.

The Mouse Test Program

```
' A DX Mouse and Button example
' GWS May 2012

def w:WINDOW
def spr1,spr2:int
def wstyle,run:int
def mx,my,wW,wH,kx,ky:int

AUTODEFINE "OFF"

wW = 600
wH = 400

wstyle = @MINBOX|@MAXBOX|@NOAUTODRAW
WINDOW w,-wW,0,wW,wH,wstyle,0,"DX Button Test",mainwindow
CONTROL w,"T,,(wW-100)/2,75,100,30,@CTEditCenter|0x200,1"
CONTROL w,"T,,(wW-100)/2,145,100,30,@CTEditCenter|0x200,2"
CONTROL w,"T,Mouse X Co-Ordinate,(wW-150)/2,55,150,15,@CTEditCenter|0x200,3"
CONTROL w,"T,Mouse Y Co-Ordinate,(wW-150)/2,125,150,15,@CTEditCenter|0x200,4"

' create a DirectX screen ...
CREATESCREEN(w,600,400)
DXFILL w, rgb(0,50,100)
centerwindow w

' load the button sprite ...
spr1 = DXSPRITE(w,GETSTARTPATH + "button.bmp",70,35,0,RGB(0,0,0))
IF spr1 = 0
   MESSAGEBOX w, "Could not load the button sprite","Error"
   CLOSEWINDOW w
   END
ENDIF

' set up the button sprite ..
DXSETSPRITEDATA spr1,@SDBLTTYPE,@BLTTRANS
DXSETSPRITEDATA spr1,@SDTRANSKEY,rgb(0,0,0)
DXMOVESPRITE spr1, (600-70)/2, 300
DXDRAWSPRITE w, spr1

' load the point sprite which will follow the cursor ..
spr2 = DXSPRITE(w,GETSTARTPATH + "point.bmp",2,2,0,RGB(0,0,0))
IF spr2 = 0
   MESSAGEBOX w, "Could not load the point sprite","Error"
   CLOSEWINDOW w
   END
ENDIF
DXSETSPRITEDATA spr2,@SDBLTTYPE,@BLTTRANS
DXSETSPRITEDATA spr2,@SDTRANSKEY,rgb(0,0,0)

SETFONT w, "Arial", 12, 600
drawmode w,@TRANSPARENT

run = 1

WAITUNTIL run = 0
CLOSEWINDOW w
END
```

(Continued)

... continuation

```
mainwindow:
SELECT @CLASS
    case @IDCLOSEWINDOW
      run = 0
    case @IDDXUPDATE
         update
      case @IDPAINT
         dxflip w
      case @IDMOUSEMOVE
' check where the cursor is, allowing for the window banner of 26px
' and the borders of 7px ..
         kx = 7 * mx / wW + 0.5      :' mousex correction for borders of 7 pixels
         mx = @mousex + kx

         ky = 26 * my / wH + 0.5     :' mousey correction window caption of 26px
         my = @mousey + ky

         setcontroltext w,1,str$(mx)
         setcontroltext w,2,str$(my)

      case @IDLBUTTONDN
' move the point sprite to the cursor and check for collision ..
         DXMOVESPRITE spr2, mx, my
         if DXHITSPRITE (spr1,spr2) then run = 0
ENDSELECT
RETURN


SUB update
' show the button text ..
move w,(600-70)/2+20,306
print w,"Exit"

'show the changes
dxflip w

RETURN
```

Run the program from the attached mouse.zip file. You will see that the mouse co-ordinates change as you move the mouse around the window.

That is the result of the **@IDMOUSEMOVE** message being intercepted in the main window messages section of the program.

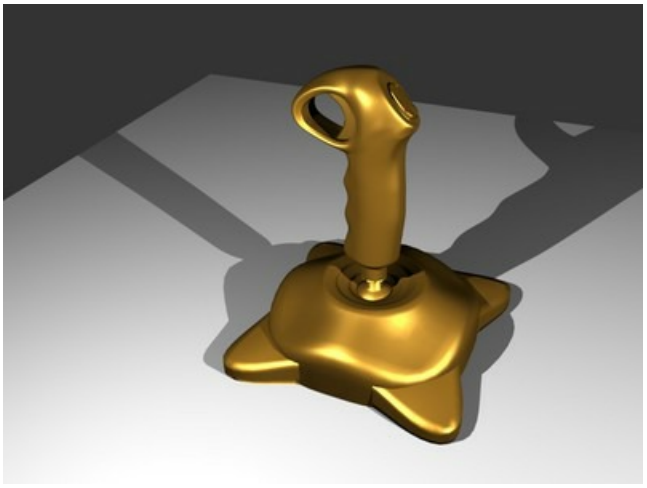
Since the mouse co-ordinates are relative only to the client part of the window, it is necessary to calculate correction factors to get the actual values, allowing for the Window caption height and the borders, as shown in the program.

The **@IDLBUTTONDN** message detects when the left mouse button is clicked.

Because we used the **@NOAUTODRAW** flag to create the window, the **@IDPAINT** message is used to redraw the screen by simply 'flipping' it again, if another window is opened on top of the test program.

That's it for mouse input - now we'll take a look at Joysticks.

Joystick Input



Joysticks are not very good pieces of equipment.

They are often poorly manufactured, plastic devices, which provide barely adequate and erratic signals to Windows.

However, it is possible to get reasonably useful signals sufficient to control a program, such as - is the user wanting to move up, down, left or right?

There are no built-in joystick commands in Creative Basic - we shall have to use the Windows API utility - **joyGetPos( ) -** to do the job.

The joystick handle can be moved in the 'x' and 'y' directions.

To obtain positional readings, it is necessary to poll the joystick, say every 200 milliseconds, using a timer.

Values are returned from the **joyGetPos( )** routine representing the 'x' and 'y' axis positions, in the range 0 to 65535. These values are usually unstable and jump about all over the place - so some smoothing is called for.

Most joysticks will also provide signals for the state of a 'trigger' and several other user buttons, and usually, the posit ion of a 'throttle' control. Sometimes the joystick handle can be twisted to provide an additional control.

To illustrate the method for setting up a program to use the joystick routines, I'll use a test program incorporating the Window's API routine - **joyGetPos( ).**

You may notice that this routine does not return data for the position of a twist handle or any point of view (POV) controls. If your program needs to access these controls, you will need to use the **joyGetPosEx( )** API function.

This is used in the same way as the **joyGetPos( )** function, but requires quite more effort to set up and use. We will take a brief look at it after the following example.

The Joystick Test Program

```
' Joystick test program
' GWS - June 2012

def win:WINDOW
def wW,wH,wstyle:int
def xmul,ymul:float
def i,run,r,x,y,Key:int
def redon1,redon2,redoff1,redoff2:int
def trigger,textw,texth,xprev,yprev:int
def pind,pind1,pind2,pind3,pval:int
def indmin,indmax,u:int
def xs,ys,xsw,ysw,centrex,centrey:int
def a$:string

autodefine "OFF"

wW = 800 : wH = 600

xmul = wW/65535
ymul = wH/65535

' UDT for joystick data ..
type joyinfo
  def Xpos:int
  def Ypos:int
  def Zpos:int
  def Button:int
endtype

def joy:joyinfo

declare "winmm.dll",joyGetPos(id:int,n:joyinfo),int

wstyle = @nocaption
window win,0,0,wW,wH,wstyle,0,"",mainwindow
setwindowcolor win,rgb(0,30,100)

centerwindow win

control win,"B, Exit,(800-70)/2, 490, 70, 40, @ctlbtnflat, 1"
control win,"T,,200,180,100,30,@cteditcenter|0x200,2"
control win,"T,,500,180,100,30,@cteditcenter|0x200,3"
control win,"T,,150,360,50,30,@cteditcenter|0x200,4"
control win,"T,,600,360,50,30,@cteditcenter|0x200,5"
control win,"T,,(wW-150)/2,430,150,30,@cteditcenter|0x200,6"

for i = 1 to 6
   setcontrolcolor win,i,rgb(250,250,250),rgb(0,90,170)
next i

' set colors for a Red light button press indicator ...
redoff1 = rgb(50,0,0): redoff2 = rgb(50,0,0)
redon1 = rgb(190,0,0): redon2 = rgb(255,0,0)

' set the maximum and minimum for the indicator bar ..
indmax = 100
indmin = 0
```

(Continued)

... continuation

```
' set the indicator limit values ...
pind1 = (wW-200)/2 + 1                  :' pixels to start of indicator
pind2 = (wW-200)/2 + 200 - 5            :' pixels to end of indicator
pind = pind2 - pind1                    :' indicator length in pixels
pind3 = indmax - indmin               :' indicator range in real values ...

trigger = 0
labels

starttimer win,200,1

run = 1

waituntil run = 0
stoptimer win,1
closewindow win
END

mainwindow:
select @class
   case @idclosewindow
    run = 0

   case @idchar
' pressing 'ESC' will exit the program ...
    Key = @code
      if (Key=27) then run = 0
' clicking the Exit button ...
   case @idcontrol
      if (@controlid = 1) then run = 0

   case @idtimer

' check joystick status ..
      r = joyGetPos(0,joy)
      x = joy.Xpos * xmul             :' in the range 0 - wW
      y = joy.Ypos * ymul             :' in the range 0 - wH

' scale x and y to the range -1000 < 0 < +1000
      xs = x*1000/wW
      ys = y*1000/wH

' maintain a weighted average for the scaled values of x and y ...
      xsw = int((4 * xsw + xs)/5)
      ysw = int((4 * ysw + ys)/5)

' display the joystick scaled values ...
      setcontroltext win,2,str$(xsw)
      setcontroltext win,3,str$(ysw)

' show a cursor lines after deleting the previous positions ...
      if (abs(xprev) >= 0) | (abs(yprev) >= 0)
' erase previous cursor line ...
      line win, xprev, 0, xprev, wH, rgb(0,30,100)
         line win, 0, yprev, wW, yprev, rgb(0,30,100)
```

(Continued)

... continuation

```
' draw the new cursor line ...
        line win, x, 0, x, wH, rgb(0,200,0)
        line win, 0, y, wW, y, rgb(0,200,0)
        xprev = x: yprev = y
' re-draw the labels ...
        labels
     endif

' display the current 'x' and 'y' pixel positon ...
     setcontroltext win,4,str$(x)
     setcontroltext win,5,str$(y)

' show the joystick direction of movement ..
     centrex = (xsw > 400) & (xsw < 600)
     centrey = (ysw > 400) & (ysw < 600)

     select 1
        case centrex & centrey
           setcontroltext win,6,"Centred"
        case (xsw <= 400)
           setcontroltext win,6,"Move Left"
        case (xsw >= 600)
           setcontroltext win,6,"Move Right"
        case (ysw >= 600)
           setcontroltext win,6,"Move Down"
        case (ysw <= 400)
           setcontroltext win,6,"Move Up"
        default
           setcontroltext win,6,""
     endselect

' check for button press ...
     if (joy.Button > 0) & (trigger = 0)
' turn on red light ...
        rect win, (800-26)/2, 49, 26, 13 ,redon1,redon1
        rect win, (800-22)/2, 50, 22, 11 ,redon2,redon2

        move win,352,68: print string$(20," "):move win,352,68
        setfont win, "Arial",8,500,0
        frontpen win, RGB(10,180,250)

        select joy.button
           case 1
              a$ = "Button 1 Pressed"
           case 2
              a$ = "Button 2 Pressed"
           case 4
              a$ = "Button 3 Pressed"
           case 8
              a$ = "Button 4 Pressed"
           case 16
              a$ = "Button 5 Pressed"
        endselect
```

(Continued)

```
            gettextsize win, a$, textW, textH
            move win,(800-textw)/2,68
            print win,a$
            trigger = 1
         endif

         if (NOT (joy.button > 0)) & (trigger = 1)
   ' turn off red light ...
            rect win, (800-26)/2,49, 26, 13 ,redoff1,redoff1
            rect win, (800-22)/2,50, 22, 11 ,redoff2,redoff2
            trigger = 0
            move win,352,65: print string$(40," "):move win,352,65
         endif

   ' display the throttle indicator pointer ...
         indic

   endselect
   Return

   Sub labels

   ' draw the red light ...
   rect win, (800-30)/2, 48, 30, 15 ,rgb(70,70,70),rgb(70,70,70)
   if (trigger = 0)
      rect win, (800-26)/2, 49, 26, 13 ,redoff1,redoff1
      rect win, (800-22)/2, 50, 22, 11 ,redoff2,redoff2
   else
      rect win, (800-26)/2, 49, 26, 13 ,redon1,redon1
      rect win, (800-22)/2, 50, 22, 11 ,redon2,redon2
   endif

   ' show the field descriptions ...
   setfont win, "Arial",12,500,0
   frontpen win, RGB(20,180,250)
   a$ = "Joystick Scaled Values"
   gettextsize win, a$, textW, textH
   move win,(800-textw)/2,100
   print win,a$

   setfont win, "Arial",10,500,0
   a$ = "Joystick Screen Co-ordinates"
   gettextsize win, a$, textW, textH
   move win,(800-textw)/2,350
   print win,a$

   setfont win, "Arial",8,500,0
   a$ = "(Values range from 0 to 1000)"
   gettextsize win, a$, textW, textH
   move win,(800-textw)/2,130
   print win,a$

   a$ = "x-axis"
   frontpen win, RGB(20,220,150)
   move win,240,185
   print win,a$
```

(Continued)

... continuation

```
a$ = "y-axis"
move win,540,185
print win,a$

setfont win, "Arial",10,500,0
frontpen win, RGB(10,180,250)
a$ = "' x ' pixels"
move win,145,340
print win,a$

a$ = "' y ' pixels"
move win,595,340
print win,a$

a$ = "Throttle"
move win,510,230
print win,a$

' draw the throttle indicator background ..
rect win,(wW-200)/2,229,200,20,0,rgb(0,90,70)

return

sub indic
' routine to set the throttle control indicator ...
u = 100 - joy.Zpos * 100/63535            : ' calculate the percentage setting
...
if (u < 0) then u = 0

a$ = str$(u) + "%"
move win,560,230:print string$(20," ")
move win,560,230
print win,a$

' calculate the pixel position of the pointer ...
pval = pind1 + (u / pind3 * pind)

' draw the throttle pointer ...
rect win, pval, 230, 5, 19 , 0, rgb(150,0,0)

return
```

I'm afraid the program grew a little bit longer than I anticipated - I added a bit more functionality - and suddenly, there it was. My programs often seem to do that.

Anyway, lets look at the main features ..

First, we declare a data structure to hold the collected joystick information ..

```
' UDT for joystick data ..

type joyinfo
  def Xpos:int
  def Ypos:int
  def Zpos:int
  def Button:int
endtype

def joy:joyinfo
```

Why do we need this ? - the Windows API **joyGetPos( )** is defined to need this data structure to hold the joystick data when it is read in.

So the 'x' co-ordinate for example, will be held in our variable **joy.Xpos**.

Next we declare the API routine to be used -

```
declare "winmm.dll",joyGetPos(id:int,n:joyinfo),int
```

Since we only retrieve data that exists at the time we make a call to the API routine, we need a timer to control how often the joystick is to be polled.

```
starttimer win,200,1
```

Our program will be notified of the current joystick values each time a call is made.

When the information is available, we get a Window's message in the **@idtimer** section of the message handling routine. All our processing of the joystick information is done in this section.

```
' check joystick status ..
      r = joyGetPos(0,joy)

      x = joy.Xpos * xmul            :' in the range 0 - wW
      y = joy.Ypos * ymul            :' in the range 0 - wH

' scale x and y to the range -1000 < 0 < +1000
      xs = x*1000/wW
      ys = y*1000/wH

' maintain a weighted average for the scaled values of x and y ...
      xsw = int((4 * xsw + xs)/5)
      ysw = int((4 * ysw + ys)/5)
```

Here we make the call to the **joyGetPos(0,joy)** API, telling it that we are using joystick 0, and to use the data structure 'joy' to collect the results.

Since the raw values will be in the inconvenient range 0 - 65535, these values are first scaled to screen size 'wW' (width) and 'wH'(height), using the scaling factors

xmul = wW / 65535, and
ymul = wH / 65535.

These values are used to draw the lines at the current 'x' and 'y' co-ordinates.

A further scaling is then done to place the ‘x,y’ co-ordinates in the range -1000 to +1000, which makes the subsequent processing logic a bit easier.

```
' scale x and y to the range -1000 < 0 < +1000
    xs = x *1000 / wW
    ys = y *1000 / wH
```

As we mentioned before, the joystick data is usually a bit unstable, the values varying intermittently, particularly around the ‘dead zone’ used for re-centring when you release the handle.

To help stabilise the values, a weighted average on incoming data is used ..

```
' maintain a weighted average for the scaled values of x and y ...
    xsw = int((4 * xsw + xs)/5)
    ysw = int((4 * ysw + ys)/5)
```

So now we have reasonably stable ‘x’ and ‘y’ values which can be used to control say, a game.

The program also shows how various joystick buttons and ‘throttle’ control values can be acted upon.

What it does not do, is provide any information about the status of a ‘twist’ handle, or a point-of-view (POV) control. That is because the **joyGetPos( )** API does not deal with that amount of detail.

If you require this information, you will need to use the **joyGetPosEx( )** API. This uses the following data structure.

```
' declare Joystick Extended Info API data structure ...
type joyinfoex
  def size:int    :' size of data structure
  def flag:int    :' flag which info to return (* see next table)
  def Xpos:int    :' x position (range 0 to 65535)
  def Ypos:int    :' y position (range 0 to 65535)
  def Zpos:int    :' z position (range 0 to 65535)
  def Rpos:int    :' rudder/4th axis position
  def Upos:int    :' 5th axis position
  def Vpos:int    :' 6th axis position
  def Buttonstate:int    :' button status
  def ButtonNumber:int    :' current button number pressed
  def POV:int       :' point of view state
  def reserve1:int    :' reserved
  def reserve2:int    :' reserved
endtype
```

Unfortunately, this API needs a lot of setting up before you can call it.

```
def JOY_POVCENTERED,JOY_POVFORWARD,JOY_POVRIGHT,JOY_POVLEFT:int
def JOY_RETURNX,JOY_RETURNY,JOY_RETURNZ,JOY_RETURNR,JOY_RETURNU,JOY_RETURNV:int
def JOY_RETURNPOV,JOY_RETURNBUTTONS,JOY_RETURNRAWDATA,JOY_RETURNPOVCTS:int
def JOY_RETURNCENTERED,JOY_USEDEADZONE,JOY_CAL_READALWAYS,JOY_CAL_READRONLY:int
def JOY_CAL_READ3,JOY_CAL_READ4,JOY_CAL_READXONLY,JOY_CAL_READYONLY:int
def JOY_CAL_READ5,JOY_CAL_READ6,JOY_CAL_READZONLY,JOY_CAL_READUONLY,JOY_CAL_READVONLY:int

def JoyStick1,JoyStick2,JOY_RETURNALL:int

' define joystick ID's ...
Const JoyStick1 = 0
Const JoyStick2 = 1

' define API constants ...
' (many of these are not used for the extended joystick info API) ..
Const JOY_POVCENTERED = -1
Const JOY_POVFORWARD = 0
Const JOY_POVRIGHT = 9000
Const JOY_POVLEFT = 27000
Const JOY_RETURNX = 1
Const JOY_RETURNY = 2
Const JOY_RETURNZ = 4
Const JOY_RETURNR = 8
Const JOY_RETURNU = 0x10
Const JOY_RETURNV = 0x20
Const JOY_RETURNPOV = 0x40
Const JOY_RETURNBUTTONS = 0x80
Const JOY_RETURNRAWDATA = 0x100
Const JOY_RETURNPOVCTS = 0x200
Const JOY_RETURNCENTERED = 0x400
Const JOY_USEDEADZONE = 0x800
def Return1,Return2,Return3:int
Return1 = (JOY_RETURNX | JOY_RETURNY | JOY_RETURNZ)
Return2 = (JOY_RETURNR | JOY_RETURNU | JOY_RETURNV)
Return3 = (JOY_RETURNPOV | JOY_RETURNBUTTONS)
Const JOY_RETURNALL = (Return1 | Return2 | Return3)
Const JOY_CAL_READALWAYS = 0x10000
Const JOY_CAL_READRONLY = 0x2000000
Const JOY_CAL_READ3 = 0x40000
Const JOY_CAL_READ4 = 0x80000
Const JOY_CAL_READXONLY = 0x100000
Const JOY_CAL_READYONLY = 0x200000
Const JOY_CAL_READ5 = 0x400000
Const JOY_CAL_READ6 = 0x800000
Const JOY_CAL_READZONLY = 0x1000000
Const JOY_CAL_READUONLY = 0x4000000
Const JOY_CAL_READVONLY = 0x8000000

def joy:joyinfoex    :' the User defined type shown previously

joy.Size = len(joy)
joy.flag = JOY_RETURNALL

declare "winmm.dll",joyGetPosEx(id:int,n:joyinfoex),int

starttimer w1,100

..

sub main
select @class
..
   case @idtimer

' check joystick status ..
      r = joyGetPosEx(0,joy)
```

Processing is then done in much the same way as our previous example.

The handle twist position for instance, is now available in **joy.rpos.**

JoyGetPosEx( ) could be used for all joystick input, but it is more cumbersome than the basic **joyGetPos( )** API.

So that's it for the hardware input / output facilities.

The next section **Input / Output - Part 2**, will deal with creating and reading disk files.