Using the Keyboard

This Chapter reveals how AOZ exploits the potential of your keyboard.
Checking for a key-press The keyboard can be used to interact with your routines once they are running. This is vital for any sort of arcade game, adventure gaming or for more practical items such as word processing.

INKEY$

function: check for a key-press

```
k$=Inkeys$
```

This function checks to see if a key has been pressed, and reports back its value in a string. For example:

```
Do
    K$=Inkey$
    If K$<>""Then Print "You pressed a key!"
Loop
```

Now use the INKEY$ function to move your cursor around the screen, like this:

```
Print "Use your cursor keys"
Do
    K$=Inkey$
    If K$<>""Then Print K$;
Loop
```

The INKEY$ function does not wait for you to input anything from the keyboard, so if a character is not entered an empty string is returned. INKEY$ can only register a key-press from one of the keys that carries its own Ascii code, and the Ascii code numbers that represent the characters which can be printed on the screen are explained in Chapter XX.X (5.2).

It has also been explained that certain keys like [Help] and the function keys [Fl] to [F10] do not carry as Ascii code at all, and if INKEY$ detects that this type of key has been pressed, a character with a value of zero will be returned. When this happens, the internal "scan codes" of these keys can be found.

SCANCODE

function: return the scancode of a key entered with INKEY$

```
s=Scancode
```

SCANCODE returns the internal scan code of a key that has already been entered using the INKEY$ function. The next example may be tested by pressing the function keys, [Del] and [Help]. To interrupt the example, press [Ctrl]+[C].

```
Do
    While K$=""
        K$=Inkey$
    Wend
    If Asc(K$)=0 Then Print "No Ascii Code"
    Print "The Scan Code is ";Scancode
    K$=""
Loop
```

SCANSHIFT

function: return shift status of key entered with INKEY$

```
s=Scanshift
```

To determine if keys are pressed at the same time as either or both of the [Shift] keys, the Scanshift function returns the following values:

Value	Meaning
0	no [Shift] key pressed
1	[Left Shift] pressed
2	[Right Shift] pressed
3	both [Shift] keys pressed

Try out the following example by pressing various keys, in combination with the [Shift] keys:

```
Do
    A$=Inkey$
    S=Scanshift
    If S<>0
        Print S
    End If
Loop
```

KEY STATE

function: test for a specific key press

```
k=Key State(scan code)
```

Use this function to check whether or not a specific key has been pressed. The relevant scan code should be enclosed in brackets, and when the associated key is being pressed KEY STATE will return a value of TRUE (-1), otherwise the result will be given as FALSE (0). For example:

```
Do
    If Key State(69)=True Then Print "ESCAPE!" : Rem Esc key pressed
    If Key State(95)=True Then Print "HELP!": Rem Help key pressed
Loop
```

KEY SHIFT

function: test the status of control keys

```
bitmap=Key Shift
```

KEY SHIFT is used to report the current status of those keys which cannot be detected by either INKEY$ or SCANCODE because they do not carry the relevant codes. These "control keys can kb tested individually, or a test can be set up for any combination of such keys pressed together. A single call to the KEY SHIFT function can test for all eventualities, by examining a bit map in the following format:

Bit	Key	Tested	Notes
0	left	[Shift]	Only one [Shift] key can be tested at a time
1	right	[Shift]	Only one [Shift] key can be tested at a time
2	[Caps Lock]		Either ON or OFF
3	[Ctrl]		
4	left [Alt]		
5	right [Alt]		
6	left [Windows]		
7	right [Windows]		

If the report reveals that a bit is set to 1, then the associated key has been held down by the user, otherwise a 0 is given. Here is a practical example:

```
Centre "Please press some Control keys"
Curs Off
Do
    Locate 14,4: Print Bin$(Key Shift,8)
Loop
```

These keys can also be used when setting up macro definitions, using the SCAN$ and KEY$ functions, and this is explained below.

CLEAR KEY

instruction: re-set the keyboard buffer

```
Clear Key
```

When an appropriate character is entered from the keyboard, its Ascii code is placed in an area of memory called the keyboard buffer. This buffer is then examined by the INKEY$ function in order to report on key presses.

CLEAR KEY completely erases this buffer and re-sets the keyboard, making it a very useful command at the beginning of a program when the keyboard buffer may be filled with unwanted information. CLEAR KEY can also be called immediately before a WAIT KEY command, to make sure that the program waits for a fresh key-press before proceeding.

Keyboard inputs

WAIT KEY

instruction: wait for a key-press

```
Wait Key
```

This simple command waits for a single key-press before acting on the next instruction. For example:

```
Print "Please press a key" : Wait key : Print "Thank you!"
```

INPUT$

function: anticipate a number of characters to input into a string

```
v$=Input$(number)
```

This function loads a given number of characters into a string variable, waiting for the user to enter each character in turn. Although characters will not appear on the screen, similar to INKEY$, the two instructions are totally different.

Here is an example:

```
Clear Key : Print "Please type in ten characters"
V$=Input$(10) : Print "You typed: ";V$
```

There is another version of INPUT$ which operates with a number of characters from a disc. Please see Chapter XX.X (10.2) for details.

INPUT

instruction: load a value into a variable

```
Input variables;
Input "Prompt string";variables;
```

The INPUT command is used to enter information into one or more variables. Any variable may be used, as well as any set of variables, providing they are separated by commas. A question mark will automatically appear at the current cursor position as a prompt for your input.

If your own "Prompt string" is included, it will be printed out before your information is entered. Please note that a semi-colon must be used between your prompt text and the variable list, a comma is not allowed for this purpose.

You may also use an optional semi-colon at the end of your variable list, to specify that the text cursor is not to be affected by the INPUT command, and will retain its original position after your data has been entered.

When INPUT is executed, the program will wait for the required information to be entered via the keyboard, and each variable in the list must be matched by a single value entered by the user. These values must be of exactly the same type as the original variables, and should be separated by commas.

For example:

```
Print "Type in a number"
Input A
Print "Your number was ";A
Input "Type in a floating point number";N#
Print "Your number was ";N#
Input "What's your name?";Name$
Locate 23, : Print "Hello ";Name$
```

LINE INPUT

instruction: input a list of variables separated by [Return]

```
Line Input variables;
Line Input "Prompt string";variables;
```

LINE INPUT is identical in usage to INPUT, except that is uses a press of the [Return] key to separate each value you enter via the keyboard instead of a comma. Try this:

```
Line Input "Type in three numbers";A,B,C
Print A,B,C
```

PUT KEY

instruction:load a string into the keyboard buffer

```
Put Key a$
```

This command loads a string of characters directly into the keyboard buffer, and it is most commonly used to set up defaults for your INPUT routines. Note that end of line returns can be included using a CHR$(13) character. In the next example, "NO" is assigned to the default INPUT string.

```
Do
    Put Key "NO"
    Input "Do you love me, Yes or No: ";A$
    B$=Upper$(A$)
    If B$="NO" Then Boom : Wait 50: Exit
Loop
```

Keyboard Macros

AOZ allows the creation of keyboard macros from the [Macros] option of the main [Editor] Menu, as detailed in Chapter 4.1. A macro is simply a command string assigned to one of the function keys, which is called up by pressing the appropriate function key and one of the [Amiga] keys together (FLNOTE: work in progress!). Once a macro has been defined, it can be used anywhere within the AMOS Professional system, and will have exactly the same effect as if the assigned commands had been entered from the keyboard. The same macro can be called from the Editor window, from Direct mode, or from inside an AOZ program.

As well as assigning macro definitions by means of the [Macro] option in the Editor, they can also be defined directly from an AOZ program using the powerful KEY$ reserved variable.

KEY$

reserved variable: define a keyboard macro

```
Key$(number)=command$
command$=Key$(number)
```

KEY$ assigns the contents of the specified command$ to a function key number from 1 to 20. Keys 1 to 10 are accessed by pressing the appropriate function key at the same time as the [left Amiga] key. Similarly, numbers 11 to 20 are accessed in conjunction with the [right Amiga] key. If these keys are not pressed simultaneously, they will be misinterpreted as two separate key presses!

Single quotes can be used to enclose a comment, which will only be displayed in your key definition list and will be completely ignored by the macro routine. For example:

```
Key$(3)="'Comment' Print"
```

Also note that by pressing [Alt]+[Quote] together, a special return code is generated.
If you need to generate a key press that has no Ascii equivalent, such as an [up arrow], the appropriate scan-code can be included in a macro definition. This is achieved by using the SCAN$ function, explained next.

SCAN$

function: return a scan-code for use with Key$

```
x$=Scan$(scan-code)
x$=Scan$(scan-code,mask)
```

The scan-code parameter refers to the scan-code of a key that is to be used in one of your macro definitions. There is also an optional mask parameter, which sets special keys such as [Ctrl] and [Alt], and the format is the same as for KEY SHIFT, explained earlier.

Improving your typing skills

The AOZ programmer is offered as much help as possible to enter listings quickly and correctly. As many structures as possible are automatically recognised and correctly formatted, even if upper and lower case is sometimes confused and spacings are not quite perfect. But even the most experienced programmer can be fumblefingered at times.

KEY SPEED

instruction: change key repeat speed.

```
Key Speed time-lag,delay-speed
```

During editing, a character or cursor movement is repeated for as long as its key is held down. This can be frustrating if it causes unwanted characters or cursor movements. KEY SPEED lets you change the repeat rate while a key is held down, to your own particular preference. State the time-lag you want to use between pressing a key and the start of the repeat sequence, measured in 50ths of a second.

Follow this by the delay-speed between each character you type, also in 50ths of a second. This line will slow everything down:

```
Key Speed 50,50: Rem One second delay
```

The following setting may well prevent you from editing at all!

```
Key Speed 1,1: Rem Ridiculously fast
```