The Bare Bones

This Chapter provides you with the bare bones that support AOZ programming.

These bones are used to build program skeletons, and you need to understand what they do and how they work before adding the lifeblood, the muscle-power and the brain-control that endow a program with its own life.

If you are an experienced programmer, you will already be familiar with these bare bones, and you can safely skip through most of this chapter.

AOZ is designed to provide you with the easiest and most convenient way of controlling all your programming needs, and even though it provides very powerful programming features, difficult concepts and terms are avoided wherever possible. This section begins with one of the simplest concepts in computing, known as "strings".

Strings

A "string" is a number of characters strung together. A set of quotation marks is placed at either end of the string to hold it together and keep it separate from the rest of the program. Each string is also identified by its own name, so that it can be called up by that name, and used elsewhere in the program. The "dollar" character $ is attached to the end of every string name, to mark the fact that this name refers to a string. On UK Keyboards, quote marks are typed in by pressing the [Shift] and [2] keys together, and the $ character is typed with [Shift] plus [4].

Characters in a string can be letters, numbers, symbols or spaces. The following example creates a simple string named A$, and it is defined by letting the name of the string equal the characters enclosed in quotes, like this:
TODO

```
A$ = "AMOS Professional"
Print A$
```

Here is another example, using three different strings:

```
A$="AMOS"
B$=""
C$="Professional"
Print A$ +B$ + C$
```

Strings are extremely useful, they can act on their own or work together, as that last example demonstrated. Try the next example now:

```
A$="AMOS PROFESSIONAL"-"S"
Print A$
```

The whole of Chapter 5.2 is devoted to how AOZ makes use of strings.

Variables

There are certain elements of a computer program that are set aside to store the results of calculations. The names of these storage locations are known as "variables".

Think of a variable as the name of a place where a value resides, and that the value can change as the result of a calculation made by your computer. Like strings, variables are given their own names, and once a name has been chosen it can be given a value, like this:

```
SCORE = 100
Print SCORE
```

That example creates a variable with the name of SCORE, and loads it with a value of 100.

Naming variables

The rules for the naming of variables are very simple. Firstly, all variable names must begin with a letter, so the following variable name is fine:

```
AOZ=1
Print AOZ
```

But the next name is not allowed:

```
2AOZ = 1
```

Secondly, you cannot begin a variable name with the letters that make up one of the AOZ command words, because this would confuse your computer. The following variable name is acceptable, because the first letters are not used by one of the AOZ commands:

```
FOOTPRINT=1
Print FOOTPRINT
```

But the next name is unacceptable, because the computer recognises the first five letters as the command PRINT:

```
PRINTFOOT=1
```

If you try and type in an illegal variable name, AOZ will spot the mistake, and point it out by splitting the illegal characters away from the rest of the name. A full list of the command words can be found in the Command Index.

Variable names can be as short as one character, and as long as 255 characters, but they can never contain a blank space. So the next name is allowed:

```
AOZ = 1
Print AOZ
```

But this is an illegal variable name:

```
AO Z =1
```

To introduce a name, or split it up, use the "underscore" character instead of spaces, by typing [Shift] and [-] together. For example:

```
_IAM A_LONG_LEGAL_VARIABLE_NAME=1
Print _IAM A_LONG_LEGAL_VARIABLE_NAME
```

Types of variables

There are three types of variable that can be used in AOZ programs.

Whole Numbers

The first of these types is where the variable represents a whole number, like 1 or 9999. These variables are perfect for holding the sort of values used in computer games, for example:

```
HI SCORE=1000000
Print HI SCORE
```

Whole numbers are called "integers", and integer variables can range from -147,483,648 up to 147,483,648.

Real number variables

Variables can also represent fractional values, such as 1.2 or 99.99 and the results from this sort of variable can be extremely accurate. The accuracy of numbers either side of a decimal point (known as "floating point" numbers) is fully explained in Chapter 5.3.

Real number variables must always have a "hash" symbol added to the end of their names, which is typed by pressing the [shift] + [3] key on English keyboards. For example:

```
REAL_NUMBER#=3.14
Print REAL_NUMBER#
```

String variables

This type of variable holds text characters, and the length of the text can be anything from zero up to 65,500 characters long. String variables are enclosed in quotation marks, and are also distinguished from number variables by a $ character on the end of their names, to tell AMOS Professional that they will contain text rather than numbers. For example:

```
NAME$="Name"
GUITAR$="Twang"
Print NAME$,GUITAR$
```

Storing variables

The original AMOS had instructions like "Set Buffer" to manage the space allocated to the variables in an AMOS program. These instructions are not necessary on AOZ and you can use as many variables as your machine allows (that is a lot).

SET BUFFER

instruction: set the size of the variable area

Has no effect in AOZ.

Arrays

It is often necessary to use a whole set of similar variables for something like a table of football results or a catalogue for a record collection. Any set of variables can be grouped together in what is known as an "array".

Supposing you have 100 titles in your record collection, and you need to tell AOZ the size of the table of variables needed for your array. There is a special command for setting up this dimension.

DIM

instruction: dimension an array

```
Dim variable name(number,number,number...)
```

The DIM command is used to dimension an array, and the variables in your record collection table could be set up with a first line like this:

```
Dim ARTIST$(99),TITLE$(99),YEAR(99),PRICE#(99)
```

Each dimension in the table is held inside round brackets, and if there is more than one element in a dimension each number must be separated from the next by a comma.

Element numbers in arrays always start from zero, so your first and last entries might contain these variables:

```
ARTIST$(0)="Aaron Copeland"
TITLE$(0)="Appalachian Spring"
YEAR(0)=1944
PRICE#(0)=12.99
ARTIST$(99)="ZZ Top"
TITLE$(99)="Afterburner"
YEAR(99)=1985
PRICE#(99)=9.95
```

To extract elements from your array, you could then add something like this to your example program:

```
Print TITLE$(0),PRICE#(0)
Print TITLE$(99),YEAR(99),PRICE#(99)
```

These tables can have as many dimensions as you like, and each dimension can have up to 65,0(K) elements. Here are some more modest examples:

```
Dim LIST(5),NUMBER#(5,5,5),WORD$(5,5)
```

Constants

Constants are a special type of number or string that can be assigned to a variable, or used in a calculation. They are given this name because their value remains constant, and does not change during the course of the program.

AOZ will normally treat all constants that are fractional numbers (floating point numbers) as whole numbers (integers), and convert them automatically, before they are used. For example:

```
A=3.141
Print A
```

Any numbers that are typed into an AOZ program are converted into a special format. When programs are listed, these numbers are converted back to their original form, and this can lead to minor discrepancies between the number that was originally typed in and the number that is displayed in the listing. There is no need to worry about this, because the value of the number always remains exactly the same.

Functions

There is a whole set of bare bones in the AOZ skeleton known as "functions". These are command words that have one thing in common: they all work with numbers in order to give a result.

FREE

function: give the amount of free memory in the variable buffer area

This function will return an always very large number (irrelevant) in AOZ.

DEF FN

structure: create a user-defined function

Def Fn name (list of variables) = expression

To create a user-defined function, give it a name and follow the name with a list of variables. These variables must be held inside a pair of round brackets, and separated from one another by commas, like these examples:

```
Def Fn NAME$(A$)=LOWER$(A$)
Def Fn X(A,B,C)=A*B*C
```

When a user-defined function is called up, my variables that are entered with it will be substituted in the appropriate positions, as demonstrated below.

FN

structure: call a user-defined function

```
Fn name(list of variables)
```

The following examples show how DEF FN is first used to define a function, and how FN calls it up:

```
Def Fn NAME$(A$,X,Y)=Mid$(A$,X,Y)
Print Fn NAME$("Professional",4,3)
Def Fn X(A,B,C)=A+B+C
Print Fn X(1,10,100)
```

The expression that equals the user-defined function can include any of the standard AOZ functions, and it is limited to a single line of a program.

Parameters

The values that are entered into an AOZ instruction are known as "parameters". If there is more than one parameter, each parameter must be separated from its neighbour by a comma.

For example, up to three parameters can be used after an INK command, in the form of various numbers which specify which colour is to be used for drawing operations, then the background colour, and the third parameter setting a border colour. So an INK command could appear like this, with its three parameters ready to draw a shape:

```
Ink 0,1,2
Bar 10,10 To 100,50
```

Any parameter can be left out, as long as its comma remains in place. When this happens, AOZ will check to see what the current value is, or if there is a default value for this parameter, and automatically assign this value to the parameter that has been omitted. For example:

```
Ink 0,1,2 : Rem Set drawing, background and border colour
Ink 3,, : Rem Set drawing colour only
Ink ,4, : Rem Set background, leave drawing and border colours alone
```

Procedures

The more complex the skeleton of a program gets, the easier it is to get lost among all of its routes and connections.

Experienced programmers usually split their programs into small units known as "procedures", which allow one aspect of the program to be tackled at a time, without getting distracted by everything else that is going on.
AOZ offers all the advantages of using procedures in the most convenient way, and Chapter XX.X is dedicated to a full explanation of how to exploit them. You will learn how each procedure module can be given its own specially defined variables and parameters, and how to take best advantage of them.

Please note that in AOZ procedures can be used within functions, and the use of the 'param' and 'param$' reserved variables are no longer necessary. For example:

```
Input "Enter two numbers"; A, B
Print "The result of A + B equals: "; ADDITION_PROC[ 1, 2 ]

Procedure ADDITION_PROC[ A, B ]
    Rem Nothing to do here...
End Proc A + B
```

Controlling a program skeleton

Once a program is running, there are a number of ways to stop it in its tracks, allowing you to control what happens
next.

WAIT

instruction: wait before performing the next instruction

Wait number of 50ths of a second

The WAIT command tells the computer to stop the program and wait for as long as you want before moving on to the next instruction. The number that follows it is the waiting time, specified in 50ths of a second.

The following example forces the program to wait for two seconds:

```
Print "I am the first instruction."
Wait 100
Print "I am the next instruction."
```

END

instruction: end the current program

End

As soon as the END command is recognised, it stops the program. Your browser will display a 'Progra ended' alert box.

```
Print "I am the first instruction."
Wait 150
End
Print "This instruction will never be executed!"
```

STOP

instruction: interrupt the current program

Stop

To stop the current program. The STOP instruction is used like this:

```
Print "Interrupt in two seconds!"
Wait 100
Stop
Print "I have been abandoned"
```

EDIT
instruction: leave current program and return to Edit Screen, if the application was launched under AOZ GUI (not yet programmer), otherwise just end the program.

```
Edit
```

Similarly, the EDIT instruction forces the program to be abandoned, and returns you straight to the Edit Screen, like this:

```
Print "Wait four seconds and then EDIT"
Wait 200
Edit
Print "I have been ignored!"
```

DIRECT

instruction: leave current program and return to Direct Mode

```
Direct
```

Use the DIRECT command to jump out of the current program and go straight to Direct Mode for testing out a programming idea (to be implemented in AOZ GUI later). If the application is not ran from the GUI, Direct just ends the program.

```
Print "Take me to Direct Mode immediately"
Direct
```

Normally, a program can be interrupted by pressing the [Ctrl] and the [C] keys together, returning you to the AOZ Edit Screen. This facility can be turned off and on at will, creating a crude sort of program protection.

BREAK OFF

BREAK ON

instructions: toggle the program break keys off and on

```
Break Off
Break On
```

The BREAK OFF command can be included in a program to stop a particular routine from being interrupted while it is running. To re-start the interrupt feature, use BREAK ON.

```
Break Off
Print "Try and press the Break keys now"
Wait 500
Break On
Print "Break keys activated"
Wait 100
Direct
Break Off
Do
    Print "Get out of that!"
    Wait Key
Loop
```

SYSTEM

instruction: this instruction used to display the Workbench screen on the Amiga. It has no effect in AOZ.

```
System
```

Separating commands in a line

So far in this Chapter, individual instructions have been separated from one another by typing them in and pressing the [Return] key to enter them on a new line of the program. In fact, the AOZ programmer will often want to place groups of related commands together on the same line of the program. This is achieved by separating your instructions with a colon character.

AOZ makes typing in instructions as simple as possible, and you will not normally have to worry about typing in correct spacings, as long as you stick to the rules. When a colon is used to split up commands, command words are recognised and given a capital letter and a space automatically. (to come)

This can be proved by typing in the next example exactly as it appears below, and hitting the Return] key:

```
Print"I'm so":wait key:print"neat!"
```

Marking the bones of a program

Imagine that the skeleton of your latest programming masterpiece is so clever and so complex that you cannot remember where everything is or what anything is supposed to do! There is a simple and effective way of marking any part of an AOZ program, by inserting typed messages to remind yourself exactly what this section of program is for.
These little comments or messages are known as "Rem statements".

AOZ supports C-like remarks as well as the original AMOS Professional remarks.

REM

structure: insert a reminder message into a program

```
Rem Typed in statement
' Typed in statement
// Typed in statement
/* Text in remarks...
continuation of text ...
... end of remark */
```

The beginning of a Rem statement is marked by REM or by the apostrophe character, which is simply a short-cut recognised by AOZ as a REM. The message or comment is then typed in from the keyboard, beginning with a capital letter. Here are some examples:

```
'An apostrophe can be used instead of the characters Rem
Rem The next line will print a greeting
Print "a greeting"
'This line is a comment that does nothing at all
Wait 75: Rem Wait one and a half seconds
'Return to the Edit Screen
Edit
```

These reminders are for human intelligence only, and when a Rem statement is encountered in a program, it is completely ignored by the computer.

Rem statements can occupy their own line, or be placed at the end of a line of the program, as long as they are separated from the last instruction by a colon. But the apostrophe character can only be used to mark a Rem statement at the beginning of a line.

The first of the next two lines is fine, but the second will create an error:

```
Print "This example is fine" : Rem Fine example
Print "Wrong!" : ' This is illegal
```