PROCEDURES

A procedure is a component of a computer program that allows the AOZ programmer to tackle one aspect of the program at a time, without becoming distracted or side-tracked by other programming considerations.

Procedures can be thought of as programming modules, each with a specific purpose and sphere of operation. This Chapter explains how procedures are created and fully exploited.

Creating a procedure

PROCEDURE

structure: create a procedure

```
Procedure NAME [list of optional parameters]
```

END PROC

structure: end a procedure

```
End Proc
End Proc return_value
```

A procedure is created in exactly the same way as a normal variable, by giving it a name. The name is then followed by a list of parameters and the procedure must be ended with an END PROC command.

PROCEDURE and END PROC commands must be placed on their own individual lines. For example:

```
Procedure HELLO
    Print "Hello, I am a procedure!"
End Proc
```

If you try and run that program, nothing will happen. This is because a procedure must be called up by name from inside your program before it can do anything. Now add the following line at the start of that last example, and then [Run] it.

```
HELLO
```

There is nothing preventing a procedure from calling itself, subject the memory available on the machine (note that in AOZ the SET BUFFER instruction has no effect).

SET STACK

instruction: set stack space

```
Set Stack number
```

This instruction is not used in AOZ and has no effect. The number of recursions when procedures call themseves is only limited by the available mamory.

PROC

structure: flag a procedure

```
Proc NAME
```

Another way to identify a procedure is to precede it with a PROC statement. Run the following example:

```
Rem Demonstrate that a procedure is being called not simply a command
Proc HELLO

Rem The same can be achieved without the Proc
HELLO
Procedure HELLO
    Print "Hey!"
End Proc
```

It is possible to place the procedure definition anywhere in your program. When AOZ encounters a procedure statement, the procedure is recognised and a jump is made to the final End Proc. In this way, there is no risk of executing your procedure by accident.

ON ... PROC

structure: trigger a jump to a procedure

```
On variable value Proc NAME
```

In this case, if a variable holds a particular value, a system is automatically triggered that forces a jump to a named procedure. Of course you can have as many values triggering off as many jumps to different procedures as you want. For example:

```
On X Proc PROCEDURE1,PROCEDURE2
```

Which is exactly the same as saying:

```
If X=1 Then PROCEDURE1
If X=2 Then PROCEDURE2
```

Normally, procedures will only return to the main program when the END PROC instruction is reached. But
supposing you need to jump out of a procedure instantly.

POP PROC

structure: leave a procedure immediately

```
Pop Proc
```

The POP PROC instruction provides you with a fast getaway, if you ever find yourself in need of escape. Try this:

```
ESCAPE
Procedure ESCAPE
    For PRISON=1 To 1000000000
    If PRISON=10 Then Pop Proc
        Print "I am abandoned."
    Next PRISON
End Proc
Print "I'm free!"
```

ON BREAK PROC

structure: jump to a procedure when break in program

```
On Break Proc NAME
```

A jump can also be made to a specified procedure when the program is interrupted. For example:

```
On Break Proc BROKEN
Do
    Print "Unbroken" : Wait 50
Loop
Procedure BROKEN
    Print "I am the procedure"
End Proc
```

Local and global variables

All of the variables that are defined inside a procedure work completely separately from any other variables in your programs. We call these variables "local" to the procedure. All local variables are automatically discarded after the procedure has finished executing, so that in the following example the same value of 1 will always be printed, no matter how many times it is called:

```
Procedure PLUS
    A=A+1 : Print A
End Proc
```

All the variables OUTSIDE of procedures are known as "global" variables, and they are not affected by any instructions inside a procedure. So it is perfectly possible to have the same variable name referring to different variables, depending on whether or not they are local or global.

When the next example is run, it can be seen that the values given to the global variables are different to those of the local variables, even though they have the same name.

Because the global variables cannot be accessed from inside the procedure, the procedure assigns a value of zero to them no mater what value they are given globally.

```
A=666 : B=999
EXAMPLE
Print A,B
Procedure EXAMPLE
    Print A,B
End Proc
```

To avoid errors, you must treat procedures as separate programs with their own sets of variables and instructions. So it is very bad practice for the AOZ to use the same variable names inside and outside a procedure, because you might well be confused into believing that completely different variables were the same, and tracking down mistakes would become a nightmare. To make life easy, there are simple methods to overcome such problems.

One method is to define a list of parameters in a procedure. This creates a group of local variables that can be loaded directly from the main program. For example:

```
Procedure HELLO[NAME$]
    Print "Hello ";NAME$
End Proc
Rem Load N$ into NAME$ and enter procedure
Input "What is your name?",N$
HELLO[N$]
Rem Load string into NAME$ and call HELLO
HELLO["nice to meet you!]
```

Note that the values to be loaded into NAME$ are entered between square brackets as part of the procedure call. This system works equally well with constants as well as variables, but although you are allowed to transfer integer, real or string variables, you may not transfer arrays by this method. If you need to enter more than one parameter, the variables must be separated by commas, like this:

```
Procedure TWINS[A,B]
Procedure TRIPLETS[X$,Y$,Z$]
```

Those procedures could be called like this:

```
TWINS[6,9]
TRIPLETS["Xenon","Yak","Zygote"]
```

SHARED

structure: define a list of global variables

```
Shared list of variables
```

There is an alternative method of passing data between a procedure and the main program. When SHARED is
placed inside a procedure definition, it takes a list of local variables separated by commas and transforms them into global variables, which can be directly accessed from the main program. Of course, if you declare any arrays as global using this technique, they must already have been dimensioned in the main program. Here is an example:

```
A=666: B=999
EXAMPLE
Print A,B
Procedure EXAMPLE
    Shared A,B
    A=B-A: B=B+1
End Proc
```

EXAMPLE can now read and write information to the global variables A and B. If you need to share an array, it should be defined as follows:

```
Shared A(),B#(),C$()
```

In a very large program, it is often convenient for different procedures to share the same set of global variables. This offers an easy way of transferring large amounts of information between your procedures.

GLOBAL

structure: declare a list of global variables for procedures

```
Global list of variables
```

GLOBAL sets up a list of variables that can be accessed from absolutely anywhere in your program. This is a simplified single command, designed to be used without the need for an explicit SHARED statement in your procedure definitions. Here is an example:

```
A=6 : B=9
Global A,B
TEST1
TEST2
Print A,B
Procedure TESTI
    A=A+1 : B=B+1
End Proc
Procedure TEST2
    A=A+B : B=B+A
End Proc
```

AOZ who are familiar with earlier versions of the AMOS system are now able to
employ the new facility of using strings in procedure definitions. As with disc names, the "wild card" characters * and ? can also be included. In this case, the * character is used to mean "match this with any list of characters in the variable name, until the next control character is reached", and the ? character means "match this with any single character in the variable name". So the next line
would define every variable as global:

```
Global "*"
```

Now look at the following example:

```
Shared A,"V*","VAR*END","A?0S*"
```

That line would declare the following variables as shared:

- A, as usual.
- Any variable beginning with the character V, followed by any other characters, or on its own.
- Any variable beginning with the letters VAR, followed by any other characters, and ending with the characters END.
- Any variable beginning with A, followed by any single letter, followed by OS, followed by any other
  characters.

GLOBAL or SHARED should be employed before the first use of the variable, otherwise it will have no effect on an interpreted program, although it will affect programs compiled with the AOZ Compiler.
Only strings may be used for this technique. Global and shared arrays cannot be defined using wild cards. These must be defined individually, using brackets. Also, if you try to use an expression in this way, an error will be generated. For example:

```
A$="AM*"
Global A$
```

In that case, the A$ variable would be regarded as global, and it would not be taken as a wild card for subsequent use.

With AOZ, you are able to define global arrays from a procedure, even if the array is not created at
root level, as follows:

```
Procedure VARIABLES
    Dim ARRAY(100,100)
    Global ARRAY()
End Proc
```

Returning values from a procedure

New in AOZ

In the original AMOS on the Amiga, procedures could not directly return values and could not be used within an expression: you had to use the PARAM keyword. AOZ treats procedures as simple functions, PARAM is no longer necessary.

How to return a parameter from a procedure? Simple indicate it after the END PROC keyword within square brackets:

```
End Proc[return_value]
```

Example of use :

```
Print JOIN_STRINGS["one","two","three"]; JOIN_STRINGS["four","five","six"]

Procedure JOIN_STRINGS[A$,B$,C$]
    Print A$,B$,C$
End Proc[A$+B$+C$]
```

Of course, AOZ is compatible with the old PARAM instructions.

PARAM

function: return a parameter from a procedure

```
Param
Param#
Param$
```

The PARAM function takes the result of an expression in an END PROC statement, and returns it to the PARAM variable. If the variable you are interested in is a string variable, the $ character is used. Also note how the pairs of square brackets are used in the next two examples:

```
JOIN_STRINGS["one","two","three"]
Print Param$

Procedure JOIN_STRINGS[A$,B$,C$]
    Print A$,B$,C$
End Proc[A$+B$+C$]
```

For real number variables, the # character must be used as in the following example:

```
JOIN_NUMBERS[1.5,2.25]
Print Param#

Procedure JOIN_NUMBERS[A#,B#]
    Print A#,B#
End Proc[A#+B#]
```

Local data statements

Any data statements defined inside your procedures are held completely separately from those in the main program. This means that each procedure can have its own individual areas of data. Let us end this Chapter with a modest example that calls the same procedure using different parameters, and then sets up additional data in variables.

```
Curs Off : Paper 0
RECORD["Francois","Lionet",29,"Genius"]
RECORD["Mel","Croucher",44,"Unemployed"]
A$="Richard" : B$="Vanner" : AGE=25 : OCC$="Slave Driver"
RECORD[A$,B$,AGE,OCC$]

Procedure RECORD[NAME$,SURNAME$,AGE,OCC$]
    Cls 0: Locate 0,3
    A$=NAME$+" "+SURNAME$
    Centre A$: Locate 0,6
    A$="Age: "+Str$(AGE)
    Centre A$: Locate 0,9
    A$="Occupation: "+OCC$
    Centre A$: Locate 0,16
    Centre "Press a key" : Wait Key
End Proc
```