Control Structures

There is a traditional group of instructions that allow computer programs to make decisions. They are usually known as control structures. This Chapter explains how AOZ takes the best of these traditions and uses them to give your computer a logical brain.

GOTO

structure: jump to a specified place in the program

```
Goto label
Goto line number
Goto expression
```

A computer program that can only obey a list of instructions one after the other is a very limited computer program indeed. One way of forcing programs to jump to specified locations is to use the old fashioned GOTO structure, followed by a target destination. In AOZ, these destinations can be a label, a line number or a variable. These are explained in Chapter XX.X.

Label markers can consist of names that use any string of letters or numbers, as -well as the underscore character "_", and they must be ended with the colon character ":" as follows:

```
Print "Jump in two seconds" : Wait 100
Goto LABEL_MARKER
Wait 180000 : Rem Wait one hour
LABEL_MARKER:
Print "Now is the time to jump!"
```

Numbers may be used to identify specific lines, and the program can be commanded to GOTO one of these
optional markers, like this:

```
Goto 5
Print "I am being ignored"
5 Print "I am line 5"
```

It should be obvious that these identification numbers have nothing to do with the number of lines in a program, but they may still lead to confusion. Labels are much easier to remember and to locate.
Expressions can also be used for this purpose, and the expression may be any string or integer. Strings hold the name of a label, and integers return a line identification number. Here is an example:

```
BEGIN:
Goto "BED"+"2"
End
BED1:
Print "This Bed will never be used"
Bed2:
Print "Welcome to Bed Two!"
Wait 20
Goto BEGIN
```

GOSUB

structure: jump to a sub-routine

```
Gosub label
Gosub number
Gosub expression
```

Packages of program instructions that perform a specific task can be thought of as "routines". When such routines are split into smaller packages of instructions, they can be thought of as "sub-routines". GOSUB is another antiquated command, and is used to perform a jump to a sub-routine. In fact, GOSUB is made redundant by the AOZ procedure system, but it can be useful for STOS users who want to convert programs.

As with GOTO, there are three alternative targets for a GOSUB instruction: labels, line numbers or expressions.

To make sub-routines easier to spot in your program listings, it is good practice to place them at the end of the main program. A statement such as EDIT or DIRECT should also be used to end the main program, which prevents AOZ from executing any GOSUBs after the main program has finished.

RETURN

instruction: return from a sub-routine called by GOSUB

```
Return
```

When a program obeys a GOSUB instruction, it must be instructed to RETURN to the main program after the subroutine has been executed. It should be noted that a single GOSUB statement can be linked to several RETURN commands, allowing exits from any number of different points in the routine, depending on the circumstances. After the RETURN, a jump is made back to the instruction immediately after the original GOSUB. For example:

```
Print "I am the main program"
For N=1 To 3
    Gosub TEST
Next N
End
TEST:
Print "Here we go GOSUB" : Wait 50
Print "Number =";N
Return
```

POP

instruction: remove RETURN information

```
Pop
```

Normally you cannot exit from a GOSUB statement using a standard GOTO, and this may be inconvenient. For
example, there could be an error that makes it unacceptable to return to the program exactly where you left it. In such circumstances, the POP command can be used to remove the return address generated by a GOSUB, allowing you to leave the sub-routine without the final RETURN statement being executed. For example:

```
Do
    Gosub THERE
Loop
HERE:
Print "I've just popped out!"
Direct : Rem No risk of accidental subroutine
THERE:
Print "Hello There!"
If Mouse Key Then Pop : Goto HERE
Return
```

Decision making

The command words used in the decision making process have very similar meanings in AOZ as they do in normal English.

IF / THEN

structure : choose between alternative actions

```
If conditions Then statements
```

The IF ... THEN structure allows simple decisions to be made within a program, so IF a condition is true THEN the computer decides to take a particular course of action. If the condition is not true, the machine does something else. For example:

```
NIGHT=12
DAY=12
Print "What time is it now?" : Wait 150
If NIGHT=DAY Then Goto BED
Print "Time I bought a watch"
Goto WATCHMAKER
BED:
Print "I think it is bed time"
WATCHMAKER:
```

*AND / _OR_*

structures: qualify a condition

```
If condition And condition Then statement
If condition Or condition Then statement
```

The list of condition; in an IF ... THEN structure can be any list of tests, including AND and OR. Try changing the conditions of the last example with either of the following lines:

```
If NIGHT=DAY And NIGHT<>12 Then Goto BED
If NIGHT<DAY Or NIGHT=12 Then Edit
```

ELSE

structure: qualify a condition

```
If condition Then statement1 Else statement2
```

ELSE is also understood when making decisions, as to what action should be taken, depending on conditions. So the last example could be changed to something like this:

```
If NIGHT+1=DAY Then Goto BED Else Shoot
```

The alternative choice of statements in this sort of structure must be a list of one or more AOZ instructions. Also remember to include a separate GOTO command if you want to jump to a label or a numbered line, otherwise the label will be treated as a procedure name and it could possibly generate an error. For example:

```
If NIGHT=1 Then Goto BED: Rem This is perfect
If NIGHT=1 Then BED: Rem This looks for a BED procedure
```

An IF ... THEN statement is limited to a single line, of a listing, which is not very satisfactory to an AOZ programmer. This technique has been superseded by a "structured test", where IF is used to trigger off a whole range of instructions, depending on the outcome of a single decision.
Structured tests

END IF

structure: terminate a structured test

```
If structured test End If
```

In a structured test, each test is set up with an IF and ended with a matching END IF, but under no circumstances can a THEN be used anywhere inside such a test! The statements in a structured test are separated by colons on any particular line, as usual, but can extend over any number of lines in your listing, as required. Look at this old fashioned schematic line:

```
If condition=true Then Goto Label1 Else Label2
```

This may now be replaced by the alternative structured test format:

```
If condition=true : Goto Label1 : Else Goto Label2 : End If
```

Here is a working example:

```
Input "Type values A,B and C: ";A,B,C
If A=B
    Print "A equals B";
Else
    Print "A is not equal to B";
    If A<>B And A<>C
        Print "or to C"
    End If
End If
```

Note how each IF statement must be paired with a single END IF to inform AOZ exactly which group of instructions is to be executed inside the test.

ELSE IF

structure: allow multiple structured tests

```
If condition Else If multiple conditions ... Else statement End If
```

This allows multiple tests to be performed. ELSE IF must be used within a normal IF ... END IF statement, and the only rule to remember is that there must be one ELSE just before the END IF. This sort of test waits for an expression, and if the expression is True, then what comes after it is executed. Here is an example:

```
If A=1
    Print "A=1"
Else If A=2
    Print "A=2"
Else If A=3
    Print "A=3"
Else
    Print "Something Else"
End If
```

If necessary, an entire test can be placed in a single line, as follows:

```
If A=1 : Print "A=1" : Else If A=2 : Print "A=2" : Else : Print "Something Else"
: End If
```

When taking logical decisions, your Amiga understands the following character symbols, which are used as a form of short-hand:

Symbol Meaning

- = equal to
- <> not equal to, != is also supported (as in C++)
- > greater than

- < less than
- = greater than or equal to
- <= less than or equal to

There are also three functions that can be called during the decision making process.

TRUE / FALSE

functions: hold value of True and False. Please note that AOZ does ot use numbers like 0 and 1 for boolean indicators, but the true Javascript boolean, "true" or "false" that are booleans not numbers.

```
value=True
value=False
```

You can use these functions in tests (IF / ELSE / END IF) or loops (REPEAT / UNTIL - FOR / NEXT etc...)

NOT

structure: toggle binary digits

```
value=Not digits
```

NOT is used to swap over every digit in a binary number from a 0 to a 1, and vice versa. For example:

```
Print Bin$(Not %11110000,8)
```

SWAP

structure: swap the contents of two variables

```
Swap a,b
Swap a#,b#
Swap a$,b$
```

Use the SWAP command to swap over the data between any two variables of the same type. For example:

```
A=10 : B=99: Print A,B
Swap A,B : Print A,B
```

Using loops

To write a separate routine for dozens of logical choices, and to end up with dozens of END IFs is not only messy, but also extremely tedious. AMOS Professional offers all of the expected programming short-cuts to allow sections of code to be repeated as often as necessary. These repeated parts of programs are known as "loops".

DO / LOOP

structure: keep repeating a list of statements

```
Do
    list of statements
Loop
```

This pair of commands will loop a list of AOZ statements forever, with DO acting as the marker position for the LOOP to return to. Both the DO and LOOP should occupy their own lines, as follows:

```
Do
    Print "FOREVER AND": Wait 25
Loop
```

EXIT

structure: break out of a loop

```
Exit
Exit number
```

EXIT forces the program to leave a loop immediately, and it can be used to escape from all the types of loop employed in AOZ, such as FOR ... NEXT, REPEAT ... UNTIL, WHILE ... WEND and DO ... LOOP.
Any number of loops may be nested inside of one another, and when used on its own, EXIT will short-circuit the innermost loop only. By including an optional number after EXIT, that number of nested loops will be taken into account before the EXIT is made, and the program will jump directly to the instruction immediately after the relevant loop. For example:

```
Do
    Do
    Input "Type in a number";X
    Print "I am the inner loop"
    If X=1 Then Exit
    If X=2 Then Exit 2
    Loop
    Print "I am the outer loop"
Loop
Print "And I am outside both loops!"
```

EXIT IF

structure: exit from a loop depending on a test

```
Exit If expression
Exit If expression,number
```

It is often necessary to leave a loop as a result of a specific set of conditions, and this can be simplified by using the EXIT IF instruction. As explained above, in conditional operations, the value -1 represents True, whereas a zero represents False. After using EXIT IF, an expression is given which consists of one or more tests in standard AOZ format. The EXIT will only be performed IF the result is found to True.

As before, an optional number can be given to specify the number of loops to be jumped from, otherwise only the current loop will be aborted. For example:

```
While L=0
    A=0
    Do
    A=A+1
    For X=0 To 100
        Exit If A=10,2 : Rem Exit from DO and FOR loops
    Next X
    Loop
    Exit 1: Rem Exit from WHILE loop
Wend
```

Conditional loops

WHILE / WEND

structure: repeat loop while condition is true

```
While condition
    list of statements
Wend
```

This pair of commands provides a convenient way of making the program repeat a group of instructions all the time a particular condition is true. WHILE marks the start of this loop, and the condition is checked for a value of true from this starting position through to the end position, which is marked by a WEND. The condition is then checked again at every turn of the loop, until it is no longer true. For example:

```
BLAZES:
Print "Please type in the number 9"
Input X
While X=9
    Cls : Print X : Wait 50 : Goto BLAZES
Wend
Print "That is not a 9!"
```

You are free to use AND, OR and NOT to qualify the conditions to be checked.

REPEAT / UNTIL

structure: repeat loop until a condition is satisfied

```
Repeat
    list of statements
Until condition
```

Unlike that last example, instead of checking if a condition is true or false at the start of a loop, the pair of commands makes its check at the end of a loop. REPEAT marks the start and UNTIL the end of the loop to be checked. This means that if a condition is false at the beginning of a WHILE ... WEND structure, that loop will never be performed at all, but if it is true at the beginning of a REPEAT
... UNTIL structure, the loop will be performed at least once. Here is an example that waits for you to press a mouse button:

```
Repeat
    Print "I can go on forever" : Wait 25
Until Mouse Key<>0
```

Controlled loops

When deciding how many times a loop is to be repeated, control can be made much more definite than relying on whether conditions are true or false.

FOR / TO / NEXT

structure: repeat loop a specific number of times

```
For index=first number To last number
    list of statements
Next index
```

This control structure is one of the programmer's classic devices. Each FOR statement must be matched by a single NEXT, and pairs of FOR ... NEXT loops can be nested inside one another. Each loop repeats a list of instructions for a specific number of times, governed by an index which counts the number of times the loop is repeated. Once inside the loop, this index can be read by the program as if it is a normal variable. Here is a simple example:

```
For X=1 To 7
    Print "SEVEN DEADLY SINS"
Next X
```

STEP

structure: control increment of index in a loop

```
For index=first number To last number Step size
```

Normally, the index counter is increased by 1 unit at every turn of a FOR ... NEXT loop. When the current value exceeds that of the last number specified, the loop is terminated. For example:

```
For DAY=1 To 365
    Print DAY
Next DAY
```

STEP is used to change the size of increase in the index value, like this:

```
For DAY=1 To 365 Step 7
    Print DAY
Next DAY
```

Forced jumps

So far, it has been explained how certain jumps are made to another part of a program by logical decisions based on whether a situation is true or false. Similar jumps can be made whenever a particular variable is recognised, in other words, regardless of any other conditions. GOTO and GOSUB are examples of a "forced" jump.

ON

structure: jump on recognising a variable

```
On variable Proc list of procedures
On variable Goto list of numbered lines or labels
On variable Gosub list of numbered lines or labels
```

ON can be used to force the program to jump to a pre-defined position when it recognises a specified variable.

Furthermore, jumps can be made to a choice of several positions, depending on what value is held by the variable at the time it is spotted. ON can force a jump to any of the following structures.
Procedures. When using an ON ... PROC structure, one or more named procedures is used as the target destination for a jump, depending on the contents currently held by a variable. Look at the following line:

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

That is exactly the same as saying:

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

It is important to note that procedures used in this way cannot include any parameters. If information is to be transferred to the procedure, it should be placed in a global variable, as explained in Chapter XX.X (5.5)

Goto is used to jump to one of a list of numbered lines, or a label, depending on the result of an expression. For example:

```
Print "Type in a value from 1 to 3"
Input X
On X Goto LABEL1,LABEL2,LABEL3
LABEL1:
Print "Ready"
LABEL2:
Print "Steady"
LABEL3:
Print "Go!"
```

For that to work properly, X must have a value from 1 up to the number of the highest possible destination. Any other values would cause problems. In fact the third line of that example is a very economical way of writing the following lines:

```
If X=1 Then Goto LABEL1
If X=2 Then Goto LABEL2
If X=3 Then Goto LABEL3
```

Now change the third line of the last example to this:

```
On X Goto LABEL3,LABEL2,LABEL1
```

Gosub. The use of an ON GOSUB structure is identical to ON ... GOTO, except that it must employ a RETURN to jump back to the instruction immediately after the ON ... GOSUB statement. Destinations may be given as the name of a label, or the identification number of a line between 1 and the maximum number of possible destinations.

ON is also used with the ON BREAK PROC structure, as well as ON ERROR GOTO, which are explained in the
relevant sections of the Procedures and Error Handling Chapters of this User Guide.

EVERY

instruction: call subroutine or procedure at regular intervals

```
Every time Gosub label
Every time Proc name
```

The EVERY statement is used to call up a sub-routine or a procedure at regular intervals, without interfering with the main program. Simply specify the length of time between every call, measured in 50ths of a second. Obviously the time taken for a sub-routine or a procedure to be completed must be less than the interval time, or an error will be generated.

After a sub-routine has been entered, the EVERY system is automatically disabled. This means that in order to call this feature continuously, an EVERY ON command must be inserted into a sub-routine before the final RETURN statement. Similarly, EVERY ON must be included in a procedure before returning to the main program with an END PROC. For example:

```
Every 50 Proc TEST
Do
    Print At(0,0); "Main Loop"
Loop
Procedure TEST
    Shared A
    Inc A: Print "This is call number ";A
    Every On
End Proc
```

EVERY ON / EVERY OFF

instruction: toggle regular EVERY calls

```
Every On
Every Off
```

As explained, EVERY ON should be used before the relevant sub-routine or procedure has finished executing. EVERY OFF is the default condition, and is used to disable the automatic calling process altogether.

Handling data

DATA

structure: place a list of data items in a program

```
Data list
```

A DATA statement lets you include whole lists of useful information in your programs. Each item in the list must be separated by a comma, like this:

```
Data 1,2,3,4
```

Also each DATA instruction must be the only statement on the current line, because anything that follows it will be ignored! Prove that with the following line:

```
Read A$: Print A$
Data "I am legal" : Print "But I am not!"
```

Data can be "read" into one or more variables, and unlike many Basic languages, AOZ allows you to include expressions as part of your data. So the following lines of code are all equally acceptable:

```
Data $FF50,$890
Data %11111111,%110011010110
Data A
Label: Data A+3/2.0-Sin(B)
Data "AMOS"+"Professional"
```

Examine those lines, and note that the A at Label will be input as the contents of variable A, and not the character A. The expression will be evaluated using the latest value of A.

Data statements may be placed at any position in your program, but any data stored inside an AOZ
procedure will not be accessible from the main program. Each procedure can have its own individual set of data statements, which are completely independent from the rest of the program. for example:

```
EXAMPLE
Read A$: Print A$
Data "I am Main Program Data"

Procedure EXAMPLE
    Read B$: Print B$
    Data "I am Procedure Data only"
End Proc
```

READ

structure: read data into a variable

```
Read list
```

When READ loads items of information that have been stored in a DATA statement into a list of variables, it uses a special marker to jump to the first item in the first DATA statement of your listing. As soon as that item of data has been read, the marker moves on to the next item in the list.

It must be remembered that the variables to be read must be of exactly the same type as the data held at the current position. If you match up one type of stored data with a different type of variable after a READ command, the appropriate error message will be given. Here is an example of correct matching:

```
N=Rnd(100)
Read A$,B,C,D$
Print A$,B,C,D$
Data "Text string",100,N,"AMOS"+"Professional"
```

RESTORE

structure: set the current READ pointer

```
Restore Label
Restore LABEL$
Restore Line
Restore number
```

To change the order in which your data is read from the order in which it was originally stored, you can alter the point where a READ operation expects to find the next DATA statement. The RESTORE command sets the position of this pointer by referring to a particular label or line number, and both labels and numbers may be calculated as part of an expression. For example:

```
Restore LAST
Read A$
Print A$
Data "First"
Data "Middle"
LAST:
Data "Last"
```

Each AOZ procedure has its own individual data pointer, so any calls to the command will apply to the current procedure only.

RESTORE is one of the AOZ programmer's most useful devices to force the computer to select information, depending on the actions of the user. It can be used for educational and business routines as well as adventure and role-playing games.