14: AMAL

If you wish to generate the smooth movement required in an arcade game, it's necessary to move each object on the screen dozens of times a second. This is a real struggle even in machine code and it's way beyond the abilities of the fastest version of Basic.

AMOS sidesteps this problem by incorporating a powerful animation language which is executed independently of your Basic programs. This is capable of generating high speed animation effects which would be impossible in standard Basic.

The **AM**OS **A**nimation **L**anguage (AMAL) is unique to AMOS Basic. It can be used to animate anything from a sprite to an entire screen at incredible speed. Up to 16 AMAL programs can be executed simultaneously using interrupts.

Each program controls the movements of a single object on the screen. Objects may be moved in complex predefined attack patterns, created from a separate editor accessory. You can also control your objects directly from the mouse or joystick if required.

The sheer versatility of the AMAL system has to be seen to be believed. Load up **1** from the MANUAL folder for a complete demonstration.

AMAL principles

AMAL is effectively just a simple version of Basic which has been carefully optimised for the maximum possible speed. As with Basic, there are instructions for program control (Jump), making decisions (If) and repeating sections of code in loops (For...Next). The real punch comes when an AMAL program is run. Not only are the commands lightning fast but all AMAL programs are **compiled** before run-time.

AMAL commands are entered using short keywords consisting of one or more capital letters. Anything in lowercase is ignored completely. This allows you to pad out your AMAL instructions into something more readable. So the M command might be entered as Move or the L instruction as Let.

AMAL instructions can be separated by practically any unused characters including spaces. You can't however, use the colon ":" for this purpose, as it's needed to define a label. We advise you use a semi-colon ";" to separate commands to avoid possible AMAL headaches.

There are two ways of creating your AMAL programs. The first is to produce your animation sequences with the AMAL accessory program and save them into a memory bank or you can define your animations inside AMOS Basic using the AMAL command. The general format of this instruction is:

```
AMAL n,a$
```

n is the identification number of your new AMAL program. As a default all programs are assigned to the relevant hardware sprite. So the first AMAL program controls sprite number one, the second sprite number two, and so on. You can change this selection at any time using a separate CHANNEL command. *a$* is a string containing a list of AMAL instructions to be performed in your program. Here's a simple example:

```
Load "AMOS_DATA:Sprites/Monkey_right.abk" : Rem Load some example sprites
Get Sprite Palette
Sprite 8,130,50,1 : Rem Place a sprite on the screen
```

```
Amal 8,"S: M 300,200,100 ; M -300,-200,100 J S" : Rem Define a small AMAL program
Amal On 8 : Rem Activate AMAL program number eight
Direct
```

The program returns you straight back to direct mode with the DIRECT command. Try typing a few Basic commands at this point. You can see the movement pattern continues regardless, without interfering with the rest of the AMOS system. Also note we have used sprite 8 to force the use of a computed sprite. All computed sprites from 8 to 15 are automatically assigned to the equivalent channel number by the AMAL system. So there's no need for any special initialisation procedures. Unless you wish to restrict the amount of hardware sprites it's safest to stick to just computed sprites in your programs. Notice how we've activated the AMAL program using the AMAL ON command. This has the format:

AMAL ON [prog]

prog is the number of a single AMAL program you wish to start. If it's omitted then all your AMAL programs will be executed at once.

AMAL tutorial

We'll now provide you with a guided tour of the AMAL system. This will allow you to slowly familiarise yourself with the mechanics of AMAL programs, without having to worry about too many technical details.

For the time being we'll be concentrating on sprite movements, but the same principles can also be applied to bob or screen animations.

Start off by loading some example sprites into memory. These can be found in the **Sprites** folder on the AMOS data disc. To get a directory of Sprite files type the following from the direct window:

```
Dir "AMOS_DATA:"
```

To load a sprite file, type a line like:

```
Load "AMOS_DATA:Sprites/octopus.abk"
```

Moving an object

As you would expect from a dedicated animation language, AMAL allows you to move your objects in a variety of different ways. The simplest of these involves the use of the Move command.

Move *(Move an object)*

M w,h,n

The M command moves an object *w* units to the right and *h* units down in exactly *n* movement steps. If the coordinates of your object were (X,Y), then the object would progressively move to X+W,Y+H.

Supposing you have a sprite at coordinates 100,100. The instruction M 100,100,100

would move it to 200,200. The speed of this motion depends on the number of movement steps. If *n* is large, then each individual sprite movement will be small and the sprite will move very slowly. Conversely, a small value for *n* results in large movement steps which jerk the sprite across the screen at high speed. Here are some examples of the move command:

```
Rem This moves an octopus sprite down the screen using AMAL
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
sprite 8,300,0,1
Amal 8,"M 0,250,50" : Amal On 8 : Wait Key

Rem This version moves an octopus sprite across the screen
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Sprite 9,150,150,1
Amal 9,"M 300,0,50" : Amal On 9 : Wait Key

Rem Moves octopus down and across the screen
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Sprite 10,150,150,1
Amal 10,"M 300,-100,50" : Amal On 10 : Wait Key

Rem Demonstrates multiple Move commands
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
M$="Move 300,0,50 ; Move -300,0,50"
Sprite 11,150,150,1
Amal 11,A$ : Amal On 11 : Wait Key
```

Notice how we've expanded M to Move in the above program. Since the letters "ove" are in lower case, they will be ignored by the AMAL system.

At first glance, Move is a powerful but unexciting little instruction. It's ideal for moving objects such as missiles, but otherwise it's pretty uninspiring.

Actually nothing could be further from the truth. That's because the parameters in the Move instruction are not limited to simple numbers. You can also use complex arithmetical expressions incorporating one of a variety of useful AMAL functions. Example:

```
Load "AMOS_DATA:Sprites\octopus.abk" : Get Sprite Palette : Sprite 12,150,150,1
Amal 12,"Move XM-X,YM-Y,32"
Amal On 12 : Wait Key
```

This smoothly moves computed sprite 12 to the current mouse position. X and Y hold the coordinates of your sprite, and XM and YM are functions returning the current coordinates of the mouse.

It's possible to exploit this effect in games like Pac-Man to make your objects *chase* the player's character. A demonstration of this procedure can be found in **2**.

The Move command can also be used to animate a whole screen. Here's a simple example:

```
Load Iff "AMOS_DATA:IFF/Frog_Screen.IFF",1
```

```
Channel 1 To Screen Display 1 : Rem Assigns AMAL program 1 to screen 1
Amal 1,"Move 0,-200,50 ; Move 0,200,50"
Amal On 1 : Direct
```

CHANNEL assigns an AMOS program to a particular object. We'll be discussing this command in detail slightly later, but the basic format is:

CHANNEL p To object n

p is the number of your AMAL program. Allowable values range from 0 to 63, although only the first 16 of these programs can be performed using interrupts.

object specifies the type of object you wish to control with your AMAL program. This is indicated using one of the following statements:

Sprite	(Values greater than seven refer to computed sprites)
Bob	(Blitter object)
Screen Display	(Used to move the screen display)
Screen Offset	(Hardware scrolling)
Screen Size	(Changes the screen size using interrupts)
Rainbow	(Animates a rainbow effect)

n is the number of the object to be animated. This object needs to be subsequently defined using the SPRITE, BOB or SCREEN OPEN instructions. Examples:

```
Channel 2 To Bob 1 : Rem Animate Bob 1 using AMAL program number 2
Channel 3 To Sprite 8 : Rem Assign channel three to a computed sprite
Channel 4 To Screen Display 0 : Rem Move default screen via AMAL
Channel 5 To Screen Offset 0 : Rem Change the screen offset within AMAL
```

Animation

Anim *(Animate an object)*

A n,(image,delay)(image,delay)...

The Anim instruction cycles an object through a sequence of images, producing a smooth animation effect. *n* is the number of times the animation cycle is to be repeated. A value of zero for this parameter will perform the animation continuously.

image specifies the number of an image to be used for each frame of your animation. *delay* determines the length of time this image is to be displayed on the screen, measured in units of a 50th of a second. Examples:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Sprite 8,260,100,1
Amal 8, "A 0,(1,2)(2,2)(3,2)(4,2)"
Amal On 8 : Direct

Load "AMOS_DATA:Sprites/Monkey_right.abk" : Get Sprite Palette
```

```
Sprite 9,150,50,11
M$="Anim 12, (1,4)(2,4)(3,4)(4,4)(5,4)(6,4) ; "
M$=M$+"Move 300,150,150 ; Move -300,-150,75"
Amal 9,M$
Amal On 9
Direct
```

The second example combines a sprite movement with an animation. Notice how we've separated the commands with a semi-colon ";". This ensures that the two operations are totally independent of each other. Once the animation sequence has been defined, AMAL will immediately jump to the next instruction, and the animation will begin.

It's important to realize that Anim only works in conjunction with sprites and bobs. So it's not possible to animate entire screens with this command.

Simple Loops

Jump *(Redirects an AMAL program)*

J label

Jump provides a simple way of moving from one part of an AMAL program to another. *label* is the target of your jump, and must have been defined elsewhere in your current program.

All AMAL labels are defined using a single uppercase letter followed by a colon. Like instructions, you can pad them out with lower case letters to improve readability. Here are some examples:

```
S:
Swoop:
Label:
```

Remember that each label is defined using just a **single** letter. So S and Swoop actually refer to the **same** label! If you attempt to define two labels starting with an identical letter, you'll be presented with a *label already defined in animation string* error.

Each AMAL program can have its own unique set of labels. Its perfectly acceptable to use the identical labels in several different programs. Example:

```
Load "AMOS_DATA:Sprites/octopus.abk"
Get Sprite Palette
Rem Set up seven computed sprites down the screen
For S=8 To 20 Step 2
   Sprite S,200,(S-7)*13+40,1
Next S
Rem Create seven AMAL programs
For S=1 To 7
   Channel S To Sprite 6+(S*2)
   M$="Anim 0,(1,2)(2,2)(3,2)(4,2) ; Label: Move "+Str$(S*2)+",0,7 ; "
   M$=M$+"Move -"+Str$((6-2)*2)+",0,7 ; Jump Label"
```

```
    Amal S,M$
Next S
Rem Okay, now animate it all!
Amal On : Direct
```

This next example repeatedly moves a sprite to the current mouse position:

```
Load "AMOS_DATA:Sprite/octopus.abk"
Get Sprite Palette
Sprite 8,150,150,1
Amal 8,"R: Move XM-X,YM-Y,8 ; Pause; Jump R"
Amal On 8
```

Since AMAL commands are performed using interrupts, infinite loops could be disastrous. So a special counter is automatically kept of the number of jumps in your program. When the counter exceeds ten, any further jumps will be totally ignored by the AMAL system.

Note: if you rely on this system, and allow your programs to loop continually, you'll waste a great deal of the Amiga's computer power. In practice, it's much more efficient to limit yourself to just a single jump per VBL. This can be achieved by adding a simple PAUSE command before each Jump in your program. See PAUSE for more details.

Variables and expressions

Let *(Assigns a value to a register)*

L register=expression

The L instruction assigns a value to an AMAL register. The action is very similar to normal Basic, except that all expressions are evaluated strictly from left to right.

Registers are integer variables used to hold the intermediate values in your AMAL programs. Allowable numbers range between -32768 to +32767. There are three basic types of register:

Internal registers

Every AMAL program has its own set of 10 *internal* registers. The names of these registers start with the letter R, followed by one of the digits from 0 to 9 (R0-R9).

Internal registers are like the local variables defined inside an AMOS Basic procedure.

External Registers

External registers are rather different because they retain their values between separate AMAL programs. This allows you to use these registers to pass information between several AMAL routines. AMAL provides you with up to 26 external registers, with names ranging from RA to RZ.

The contents of any internal or external register can be accessed directly from your Basic program using the AMREG function (explained later).

Special Registers

Special registers are a set of three values which determine the status of your object.

X, Y contain the coordinates of your object. By changing these registers you can move your object around on the screen. Example:

```
Load "AMOS_DATA:Sprites/Frog_Sprites.abk" : Channel 1 To Bob 1
Flash Off : Get Sprite palette : Bob 1,0,0,1
Amal 1,"Loop: Let X=X+1 ; Let Y=Y+1; Pause; Jump Loop"
Amal On 1 : Direct
```

A stores the number of the image which is displayed by a sprite or bob. You can alter this value to generate your own animation sequences like so:

```
Load "AMOS_DATA:Sprites/Frog_Sprites.abk" : Get Sprite Palette : Flash Off
Channel 2 to Bob 1 : Bob 1,300,100,1
M$="Loop: Let A=A+1 ; "
M$=M$+"For R0=1 To 5 ; Next R0 ; Jump Loop"
Amal 2,M$
Amal On 2 : Direct
```

The *For To Next* loop will be explained in more detail below. It is used here to slow down each change to Bob 1's image. When the *Next* of the loop is executed, AMAL won't continue until a vertical blank has occurred. Also note the use of ";" to separate the AMAL instructions – although a space will serve just as well.

Operators

AMAL expressions can include all the normal arithmetic operations, except MOD. You can also use the following logical operations in your calculations:

&	logical AND
|	logical OR

Note that it's not possible to change the order of evaluation using brackets "()" as this would slow down your calculations considerably and thus reduce the allowable time in the interrupt. Now for some more examples for you to type in:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Hide
Get Sprite Palette
Sprite 8,X MOUSE,Y MOUSE,1
Amal 8,"Loop: Let X=XM ; Let Y=YM ; Pause ; Jump Loop"
Amal On 8
```

```
Load "AMOS_DATA:Sprites/octopus.abk" : Hide
Get Sprite Palette
Sprite 8,X MOUSE,Y MOUSE,1
Amal 8,"Anim 0,(1,4)(2,4)(3,4)(4,4) ; Loop: Let X=XM ; Let Y=YM ; Pause ; Jump Loop"
Amal On
```

The above examples effectively mimic the CHANGE MOUSE command. However this system is much more powerful as you can easily move bobs, computed sprites, or even screens using exactly the same technique.

Making Decisions

If *(Branch within an AMAL string)*

If test Jump L

This instruction allows you to perform simple tests in your AMAL programs. If the expression *test* is -1 (true) the program will jump to label *L*, otherwise AMAL will immediately progress to the next instruction. Note that unlike it's basic equivalent, you're limited to a single jump operation after the test.

It's common practice to pad out this instruction with lowercase commands like "then" or "else". This makes the action of the command rather more obvious. Here's an example:

```
If X>100 then Jump Label else Let X=X+1
```

test can be any logical expression you like, and may include:

<>	Not equals
<	Less than
>	Greater than
=	Equals

Example:

```
Load "AMOS_DATA:Sprites/octopus.abk"
Get Sprite Palette
Sprite 8,130,50,1
C$="Main: If XM>100 Jump Test: "
C$=C$+" Let X=XM"
C$=C$+" Test: If YM>100 Jump Main "
C$=C$+" Let Y=YM Jump Main"
Amal 8,C$ : Amal On : Direct
```

A larger example can be found in **4** which allows you to control the position of a sprite using the joystick. This is actually quite crude and could be speeded up dramatically with the help of the AUTOTEST command. See AUTOTEST.

Warning! Don't try to combine several tests into a single AMAL expression using "&" or "|". Since expressions are evaluated from left to right, this will generate an error. Take the expression: X>100|Y>100. This is intended to check whether X>100 OR Y>100. In practice, the expression will be evaluated in the following order:

X>100	May be TRUE or FALSE
|Y	OR result with Y

>100 Check if (Y>100|Y)>100)

The result from the above expression will obviously bear no relation to the expected value. Technically-minded users can avoid this problem by using boolean algebra. First assign each test to an single AMAL register like so:

```
Let R0=X>100; Let R1=Y>100
```

Now combine these tests into a single expression using "|" and "&" and use it directly in your If statement.

```
If R0 | R1 then Jump L ...
```

This may look a little crazy, but it works beautifully in practice.

For To Next *(Loop within AMAL)*

For reg=start To end
: :
Next reg

This implements a standard FOR...NEXT loop which is almost identical to its Basic equivalent. These loops can be exploited in your programs to move objects in complex visual patterns. *reg* may be any normal AMAL register (R0-R9 or RA-RZ). However you can't use special registers for this purpose.

As with Basic, the register after the Next must match with the counter you specified in the For, otherwise you'll get an AMAL syntax error. Also note that the step size is always set to one. Additionally, it's possible to "nest" any number of loops inside each other.

Note that each animation channel will only perform a single loop per VBL. This synchronizes the effects of your loops with the screen display, and avoids the need to add an explicit Pause command before each Next.

Generating an attack wave for a game

These loops can be used to create some quite complex movement patterns. The easiest type of motion is in a straight line. This can be generated using a single For...Next loop like so:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Sprite 8,130,60,1
C$="For R0=1 To 320 ; Let X=X+1 ; Next R0" : Rem Move Sprite from left to right
Amal 8,C$ : Amal On 8 : Direct
```

You can now expand this program to sweep the object back and forth across the screen.

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Sprite 8,130,60,1
C$="Loop: For R0=1 to 320 ; Let X=X+1 ; Next R0 ; " : Rem Move sprite forward
```

```
C$=C$+"For R0=1 To 320 ; Let X=X-1 ; Next R0 ; Jump Loop" : Rem Move Sprite back
Amal 8,C$ : Amal On 8 : Direct
```

The first loop moves the object from left to right, and the second from right to left.

So far the pattern has been restricted to just horizontal movements. In order to create a realistic attack wave, it's necessary to incorporate a vertical component to this motion as well. This can be achieved by enclosing your program with yet another loop.

```
Load "AMOS_DATA:Sprites/octopus.abk"
Get Sprite Palette
Sprite 8,130,60,1
C$="For R1=0 To 10 ;"
C$=C$+"For R0=1 To 320 ; Let X=X+1 ; Next R0 ;" : Rem Move forward
C$=C$+"Let Y=Y+8 ; " : Rem Move Sprite down screen
C$=C$+"For R0=1 To 320 ; Let X=X-1 ; Next R0 ;" Rem Move back
C$=C$+"Let Y=Y+8 ; Next R1":Rem Move Sprite down
Amal 8,C$:Amal On 8
```

The above program generates a smooth but quite basic attack pattern. A further demonstration can be found in **Example 14.1** in the MANUAL folder.

Recording a complex movement sequence

PLay

PLay path

If you've looked at the smooth attack waves in a modern arcade game, and thought them forever beyond your reach, think again. The AMAL Play command allows you freely animate your objects through practically any sequence of movements you can imagine. It works by playing a previously defined movement pattern stored in the AMAL memory bank.

These patterns are created from the AMAL accessory on the AMOS program disc. This simply records a sequence of mouse movements and enters them directly into the AMAL memory bank. Once you've defined your patterns in this way, you can effortlessly assign them to any object on the screen, reproducing your original patterns perfectly. Both the speed and the direction of your movement can be changed at any time from your AMOS Basic program.

The first time AMAL encounters a play command, it checks the AMAL bank to find the recorded movement you specified using the *path* parameter. *path* is simply a number ranging from one to the maximum number of patterns in the bank. If a problem crops up during this phase, AMAL will abort the play instruction completely, and will skip to the next instruction in your animation string.

After the pattern has been initialised, register R0 will be loaded with the tempo of the movement. This determines the time interval between each individual movement step. All timings are measured in units of a 50th of a second. By changing this register within your AMAL program, you can speed up or slow down your object movements accordingly.

Note that each movement step is **added** to the current coordinates of your object. So if an object is subsequently moved using the Sprite or Bob instructions, it will continue it's

manoeuvres unaffected, starting from the new screen position. It's therefore possible to animate dozens of different objects on the screen using a single sequence of movements.

Register R1 now contains the a flag which sets the direction of your movements. There are three possible situations:

• R1>0 Forward

A value of one for R1 specifies that the movement pattern will be replayed from start to finish, in exactly the order it was created. (Default)

• R1=0 Backward

Many animation sequences require your objects to move back and forth across the screen in a complex pattern. To change direction, simply load R1 with a zero. Your object will now turn around and execute your original movement steps in reverse.

• R1=-1 Exit

If a collision has been detected from your AMOS program, you'll need to stop your object completely, and generate an explosion effect. This can be accomplished by setting R1 to a value of minus one. AMAL will now abort the play instruction, and immediately jump to the next instruction in your animation sequence.

The clever thing about these registers is that they can be changed directly from AMOS Basic. This lets you control your movement patterns directly from within your main program. There's even a special AMPLAY instruction to make things easier for you.

The PLay command is perfect for controlling the aliens in an arcade game. In fact, it's the single most powerful instruction in AMAL.

AMAL *(Call an AMAL program)*

AMAL n,a$
AMAL n,p
AMAL n,a$ to address

The AMAL command assigns an AMAL program to an animation channel. This program can be taken either from a string in *a$* or directly from the AMAL bank.

The first version of the instruction loads your program from the string *a$* and assigns it to channel *n*. *a$* can contain any list of AMAL instructions. Alternatively you can load your program from a memory bank created using the AMAL accessory. *p* now refers to the number of an AMAL program stored in bank number 4.

n is the number of an animation channel ranging from 0 to 63. Each AMOS channel can be independently assigned to either a bob, a sprite or a screen.

Only the first 16 AMAL programs can be performed using interrupts. In order to exceed this limit you need execute your programs directly from Basic using the SYNCHRO command.

The final version of the AMAL instruction is provided for advanced users. Instead of moving an actual object, this simply copies the contents of registers X,Y and A into a specific area of memory. You can now use this information directly in your own Basic

routines. It's therefore possible to exploit the AMAL system to animate anything from a BLOCK to a character. The format is:

AMAL n, a$ To address

address must be EVEN and must point to a safe region of memory, preferably in an AMOS string or a memory bank. Every time your AMAL program is executed (50 times per second), the following values will be written into this memory area:

Location	Effect
Address	Bit 0 is set to 1 If the X has changed. Bit 1 indicates that Y has been altered. Bit 2 will be set if the image (A) has changed since the last interrupt.
Address+2	Is a **word** containing the latest value of X
Address+4	Holds the current value of Y
Address+6	Stores the value of A.

These values can be accessed from your program using a simple DEEK.
Note: This option totally overrides any previous CHANNEL assignments.

AMAL Commands

Here is a full list of the available AMAL commands:

Move

This moves an object smoothly from one position to another. The syntax is:

Move deltaX, deltaY, steps

deltaX holds the distance to be moved horizontally. Positive numbers indicate a movement from left to right, and negative values from right to left.

deltaY specifies the vertical displacement. If *deltaY* is positive then your object will move down the screen, otherwise it will drift upwards.

n indicates the number of steps the movement is to be performed in. The smoothest movements are generated when both *deltaX* and *deltaY* are exact multiples of *n.*

A (Anim)

Anim cycles,(image,delay)(image,delay)...

The Anim instruction assigns a sequence of images to either a sprite or a blitter object to generate a realistic animation effect.

cycles specify the number of times the animation is to be repeated. If it's set to zero, the animation will continue indefinitely. *image* chooses the image number for each frame of your animation. *delay* sets the amount of time (in 50ths of a second) the image will be displayed.

After the Anim command has been initialised, AMAL will automatically jump to the next

instruction. This allows you to combine both animation and movement in the same AMAL program.

Let

Let reg=exp

This command assigns a value to an AMAL register. *reg* is the name of the AMAL register to be changed. There are 10 internal registers ranging from R0 to R9 available for your use, and a further 26 external registers (RA to RZ). You can also alter the position and type of your object directly using the special registers X,Y and A.

expr is a standard arithmetical expression and is evaluated from left to right to produce the final result.

Most of the normal operators are supported including +,-,* and /. However you are not allowed to change the order of calculation using brackets "()".

Jump

Jump L

The Jump command jumps from the current point in your AMAL program to label *L*. *L* is the name of a label which has been previously defined in your AMAL string. Labels consist of single capital letter and are created using a ":" as in standard Basic.

If

If exp Jump L

The If instruction allows you to jump from one part of an AMAL program to another depending on the result of a test. *exp* is a logical expression in the standard format

If *exp* is TRUE then the program will jump to label *L*, otherwise it will immediately execute the next instruction after the Jump.

There are two other forms of this command which are used by the AUTOTEST feature:

If exp Direct L (Chooses part of program to be executed after an autotest)
If exp eXit (Leaves Autotest)

See AUTOTEST for more information.

For To Next

For Reg=start To end ...Next Reg

This is a direct implementation of Basic's FOR...NEXT loops. *Reg* can be any internal or external AMAL register. As normal, loops can be nested but the step size of your loop is always set to one.

Note that AMAL will automatically wait for the next vertical blank before jumping back to the start of your loop with Next. Since the object movements in your program will only

be seen after the screen is updated after the VBL, faster loops would simply waste valuable processor time with no visible effect. So your For...Next loops are automatically synchronized with the screen updates to produce the smoothest possible results.

PLay

PLay path

The PL command animates your objects through a series of movements stored in the AMAL bank. These patterns are entered directly with the mouse, using the powerful AMAL accessory utility. So there's no real limit to the type of patterns you can produce with this system.

path is the number of a pattern which has been previously saved in the AMAL bank. If this pattern does not exist, AMAL will skip the PL instruction, and immediately jump to the next command in your animation sequence.

All movements are performed relative to the current position of your objects. It's therefore possible to move an entire attack wave using a single path definition. You can also move an object directly from Basic without affecting the movement in the slightest. The status of the current movement is controlled through two AMAL registers.

R0 holds the tempo of your movement. Increasing this value will speed up the object on the screen.

R1 contains the direction of the motion. There are three possible alternatives.

R1>0 Moves through the movement sequence in the original order .

R1=0 Executes your movement steps in reverse.

R1=-1 Stops the movement sequence completely and proceeds to the next AMAL instruction.

The contents of these registers can be changed at any time from within your Basic program using either the AMREG or the special AMPLAY command.

A further explanation of this instruction can be found in the AMAL tutorial near the beginning of this chapter. Also see **Example 14.2** in the MANUAL folder.

Warning: It is essential that you use semi-colons to split up your AMAL instructions. The following string will generate an *AMAL bank not reserved* error simply because there is no separator.

```
A$="Pause Let R0=1"
```

The correct syntax is:

```
A$="Pause ; Let R0=1"
```

End

End

Terminates the entire AMAL program and turns off the Autotest feature if it's been defined.

Pause

Pause

Pause temporarily halts the execution of your AMAL program and waits for the next vertical blank period. After the VBL your program will be automatically resumed starting from the next instruction.

Pause is often used before a Jump command to ensure that the number of jumps is less than the maximum of 10 per VBL. This frees valuable processor time for your Basic programs, and can have a dramatic effect on their overall speed. So try to get into the habit of preceding your Jump commands with a Pause instruction as it's much more efficient.

AUtotest

AU (List of tests)

The Autotest feature of AMAL has been designed to provide fast interaction between AMAL and the user. It adds a special test at the start of the AMAL program which is performed every VBL before the rest of the AMAL program is executed. See the Autotest system for more details.

e**X**it

eXit

Exits from an Autótest and re-enters the current AMAL program.

Wait

Wait

Wait freezes your AMAL program and only executes the Autotest.

On

On

ON activates the main program after a wait command.

Direct

Direct

Sets the section of the main program to be executed after an autotest.

AMAL functions

=XM *(Returns the X coordinate of the mouse)*

This function is exactly the same as the X MOUSE function in AMOS Basic. It returns the X coordinate of mouse cursor in hardware coordinates.

=YM *(Returns the Y coordinate of the mouse)*

YM returns the Y coordinate of the mouse pointer as a hardware coordinate.

=K1 *(Status of left mouse key)*

K1 returns a value of -1 (true) if the left mouse key has been pressed, otherwise 0 (false).

=K2 *(Status of right mouse key)*

Returns the state of the right mouse button. If the button has been pressed then K2 will return -1 (true).

=J0 *(Tests right joystick)*

The J0 function tests the right joystick and returns a bit-map containing the current status. See JOY for more details.

=J1 *(Test left joystick)*

This tests the left joystick and returns a bit-pattern in standard format.

=Z(n) *(Random number)*

The Z function returns a random number from -32767 to 32768. This number can be limited to a specific range using the bit-mask *n*.

A logical AND operation is performed between the bit mask *n* and the random number to generate the final result. So setting *n* to a value of 255 will ensure that the numbers will be returned in the range 0 to 255.

Since this function has been optimized for speed, the number returned isn't totally random. If you need really random numbers, you would be better to generate your values using Basic's RND and then load them into an external AMAL register with the AMREG function.

=XH *(Convert a screen x coordinate into a hardware coordinate)*

=XH(s,x)

This converts a screen *x* coordinate into its equivalent hardware coordinate relative to screen *s*.

=YH *(Converts a screen Y coordinate into hardware format)*

=YH(s,y)

YH transforms a *y* coordinate from screen format into hardware format relative to screen *s*.

=XS *(Hardware to screen conversion)*

=XS(s,x)

Changes hardware coordinate *x* into a graphic coordinate relative to screen *s*.

=YS *(Hardware to screen conversion)*

=YS(s,y)

Transform hardware *y* coordinate into its equivalent screen coordinate.

=BC *(Check for collisions between bobs)*

=Bob Col(n,s,e)

BC is identical to the equivalent AMOS Basic BOB COL instruction. It checks bob number *n* for collisions between bobs *s* to *e*.

If a collision has been detected, then BC will return a value of -1 (true), otherwise 0 (false). This instruction may **not** be performed within an interrupt. So it's only available when you are executing you AMAL routines directly from Basic with the SYNCHRO instruction.

=SC(m,s,e) *(Sprite Collisions)*

=Sprite Col(n,s,e)

This is equivalent to the SPRITE COL function in AMOS Basic. It checks sprite *n* for collisions between sprites *s* to *e*. If the test is successful, a value of -1 (true) will be returned. Like the previous BC function it is only allowed in conjunction with the SYNCHRO instruction.

=C(n) *(Col)*

Returns the status of object *n* after an SC or BC function. If the object has collided then this function will return a value of -1 (true), otherwise 0 (false).

=V(v) *(Vumeter)*

The VU function samples one of the sound channels and returns the intensity of the current

voice. This is a number in the range 0-255. You can use this information to animate your objects in time to the music. An example of this can be found in **Example 14.3.** Also see the VUMETER function from AMOS Basic.

Controlling AMAL from Basic

AMAL ON/OFF *(Start/stop an AMAL program)*

AMAL ON [n]

Once you've defined your AMAL program you need to execute it using the AMAL ON command. This activates the AMAL system and starts your programs from the first instruction.

AMAL ON activates all your programs. The optional parameter *n* allows you to start just one routine at a time.

AMAL OFF [n]

Stops one or all AMAL programs from executing. These programs are erased from memory. They can only be restarted by redefining them again using the AMAL instruction.

AMAL FREEZE *(Temporarily freeze an AMAL program)*

AMAL Freeze [n]

Stops one or more AMAL programs from running. Your programs can be restarted at any time using a simple call to AMAL ON. Note that this instruction should always be used to stop AMAL before a command such as DIR is executed, otherwise problems with timing can cause visual mishaps.

=AMREG= *(Get the value of an external AMAL register)*

r=AMREG(n, [channel])
AMREG(n, [channel])=expression

The AMREG function allows you to access the contents of internal and external AMAL register directly from within your Basic program.

n is the number of the register. Possible values range from 0 to 25 with zero representing register RA and twenty-five denoting RZ.

By using the optional *channel* parameter you can reference any AMAL internal register. In this mode *n* ranges between 0 and 9 representing R0 to R9.

The following example shows how it is possible to retrieve a sprite's current X position from Basic:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Channel 1 To Sprite 8 : Sprite 8,100,100,1
A$="Loop: Let RX=X+1; Let X=RX; Pause; Jump Loop" : Rem X will overflow when >640
```

```
Amal 1,A$ : Amal On : Curs Off
Do
   Locate 0,0
   Z=Asc("X")-65 : Rem Note the use of ASC to get the register number
   Print Amreg(Asc("X")-65)
Loop
```

AMPLAY *(Control an animation produced with PLay)*

AMPLAY tempo,direction[start TO end)]

Any movement sequences you've produced using the AMAL PL command are controlled through the internal registers R0 and R1. Each object will be assigned it's own unique set of AMAL registers. So if you're animating several objects, you'll often need to load a number of these registers with exactly the same values.

Although this can be achieved using the standard AMREG function, it would obviously be much easier if there was a single instruction which allowed you to change R0 and R1 for a whole batch of objects at a time. That's the purpose of the AMPLAY command.

AMPLAY takes the *tempo* and *direction* of your movements, and loads them into the registers R0 and R1 in the selected channels.

tempo controls the speed of your object. on the screen. It sets a delay (in 50ths of a second) between each successive movement step.

direction changes the direction of the motion Here's a list of the various different options.

Value	Direction of motion
>0	Move the selected object in the original movement direction.
0	Reverses the motion and moves the object backwards.
-1	Aborts movement pattern and jumps to the following instruction in your AMAL animation sequence.

As a default, this instruction will affect all current animation channels. This can be changed by adding some explicit *start* and *end* points to the command. *start* is the channel number of the first object to be adjusted. *end* holds the channel number assigned to the last object in your list.

Note that either the *tempo* or the *direction* can be omitted as required. Examples:

```
Amplay ,0 : Rem Reverse your objects
Amplay 2, : Rem Slow down your movement patterns
Amplay 3,1 : Rem Set temp to three and direction to 1
Amplay ,-1 3 To 6 : Rem Stop movements on channels 3,4,5 and 6
```

=CHANAN *(Test Amal animation)*

s=CHANAN(channel)

This is a simple function which checks the status of an AMAL animation sequence and returns -1 (true) if it is currently active or 0 (false) if the animation is complete. *channel* holds the number of the channel to be tested. Here's an example:

```
Load "AMOS_DATA:Sprites/Monkey_right.abk" : Get Sprite Palette
Sprite 9,150,150,11
M$="Anim 12,(11,4)(12,4)(13,4)(14,4)(15,4)(16,4);"
Amal 9,M$ : Amal On
While Chanan(9)
Wend
Print "Animation complete"
```

=CHANMV *(Checks whether an object is still moving)*

s=CHANMV(channel)

Returns a value of -1 (true) if the object assigned to *channel* is currently moving, otherwise 0 (false).
This command can be used in conjunction with the AMAL Move instruction to check whether a movement sequence has "run out" of steps. You can now restart the sequence at the new position with an appropriate movement string if required. Example:

```
Load "AMOS_DATA:Sprites/Monkey_right.abk" : Get Sprite palette
Sprite 9,150,50,11
M$="Move 300,150,150; Move -300,-150,75"
Amal 9,M$ : Amal On
While Chanmv(9)
Wend
Print "Movement complete"
```

AMAL errors

=AMALERR *(Return the position of an error)*

p=AMALERR

AMALERR returns the position in the current animation string where an error has occurred. Careful inspection of this string will allow you to quickly correct your mistakes. Example:

```
Load "AMOS_DATA:Sprites/Octopus.abk"
Sprite 8,100,100,1
A$="L: IF X=300 then Jump L else X=X+1; Jump L"
Amal 8,A$
```

This program will generate a syntax error because IF will be interpreted as the two instructions "I" and "F". To find the position in the animation string of this error, type the following instruction from the direct window.

```
Print Mid$(A$,Amalerr,Amaller+5)
```

Error messages

If you make a mistake in one of your AMAL programs, AMOS will exit back to Basic with an appropriate error message. Here's a full list of the errors which can be generated by this system, along with an explanation of their most likely causes.

Bank not reserved: This error is caused if you attempt to call the PLay instruction without first loading a bank containing the movement data into memory. This should be created with the AMAL accessory program. If you not using PLay at all then check that you've correctly separated any Pause and **L**et instructions in your program.

Instruction only valid in autotest: You've inadvertently called either the Direct or the eXit instructions from your main AMAL program.

Illegal instruction in Autotest: Autotest may only be used in conjunction with a limited range of AMAL commands. It's not possible to move or animate your objects in any way inside an autotest. So check for erroneous commands like Move, Anim or For..Next.

Jump To/Within Autotest in animation string: The commands inside an autoest function are completely separate from your main AMAL program. So AMAL does not allow you to jump directly inside an AUtotest procedure. To leave an autoest,and return to your main AMAL program you must use either eXit or Direct .

Label already defined in animation string: You've attempted to define the same label twice in your AMAL program. All AMAL labels consist of just a single CAPITAL letter. So **Test** and **Total** just different versions of the same label (**T**). This error is also generated if you have accidentally separated two instructions by a ":" (colon). Use a semi-colon instead.

Label not defined in animation string: This error is generated when you try to jump to a label which does not currently exist in your animation string.

Next without For in animation string: Like it's Basic equivalent each For command should be matched by a corresponding Next. statement. Check any nested loops for an spurious Next command.

Syntax error in animation string: You've made a typing mistake in one of your animation strings. It's easy to cause this error by accidentally entering an AMAL instruction in full, just like it's Basic equivalent.. Remember that AMAL commands only consist of one or letters CAPITALS. So If you attempt to type instructions like:FOR or NEXT you'll get an error. The correct syntax of these commands are For..Next

Animation channels

AMOS allows you to execute up to 64 different AMAL programs simultaneously. Each program is assigned to a specific animation *channel.*

Only the first 16 channels can be performed using interrupts. If you need to animate more objects you'll have to turn off the interrupts using SYNCHRO OFF. You can now execute the AMAL programs step by step using an explicit call to the SYNCHRO command in your main program loop. As a default, all interrupt channels are assigned to the relevant hardware sprite.

CHANNEL *(Assign an object to an AMAL channel)*

CHANNEL n TO object s

The CHANNEL command assigns an animation channel to a particular screen related *object.* In AMAL, you're not restricted to a single channel per object. however. Any single screen object can be safely animated with several channels if required. There are various different forms of this instruction.

Animating a computed sprite

CHANNEL n TO SPRITE s

This assigns sprite number *s* to channel *n.* As a default, channels from 0 to 7 are automatically allocated to the equivalent hardware sprite, and 8 to 15 are reserved for the appropriate computed sprites.

In order to animate the computed sprites from 16 onwards, you'll need to allocate them directly to an animation channel with the CHANNEL command. As normal, sprite numbers from 8 to 63 specify a computed sprite rather than a single hardware sprite. For example:

```
Channel 5 To Sprite 8:Rem Animates computed sprite 8 using channel 5.
```

The X,Y registers in your AMAL program now refer to the hardware coordinates of the selected sprite. Similarly the current sprite image is held in register A.

Animating a blitter object

AMAL programs can also be used to animate blitter objects.

CHANNEL n TO BOB b

Allocates blitter object *b* to animation channel *n.* This object will be treated in an identical way to the equivalent hardware sprite. The only difference is that registers X and Y now contain the position of your bob in **screen** coordinates.

Note that if you've activated screen switching with the DOUBLE BUFFER command, this will be automatically used for all bob animations. For a complete example see **8** from the MANUAL folder.

Moving a screen

AMOS Basic allows you to freely position the current screen anywhere on your TV display. Normally this is controlled with the SCREEN DISPLAY instruction. However, sometimes it's useful to be able to move the screen using interrupts.

CHANNEL n TO SCREEN DISPLAY d

This sets the channel *n* to screen number *d*. Screen *d* can be defined anywhere in your program. You'll only get an error if the screen hasn't been opened when you start your animation.

The X and Y variables in AMAL now hold the position of your screen in hardware coordinates. Register A is **not** used by this option and you can't animate screens using Anim. Otherwise all standard AMAL instructions can be performed as normal. So you can easily use this system to "bounce" the picture around the display.Examples:

```
Load Iff "AMOS_DATA : IFF/Frog_screen.IFF",1
Channel 0 To screen display 1
Amal 0,"Loop: Move 0,200,100 ; Move 0,-200,100 ; Jump Loop"
Amal on 0 : Direct

Load Iff "AMOS_DATA : IFF/Frog_screen.IFF",1
Channel 0 to screen display 1
Rem Screen can only be displayed at certain postions in the X
Amal 0,"Loop: Let X=XM; Let Y=YM; Pause; Jump Loop"
Amal On : Direct
```

For a further example of this technique, load **Example 14.4** from the MANUAL folder. This demonstrates how the SCREEN DISPLAY can be used in conjunction with the menu commands to slide the menu screen up and down your display. It's similar to the display system found in Magnetic Scrolls' excellent series of adventures.

Hardware Scrolling

Although hardware scrolling can be performed using AMOS Basic's SCREEN OFFSET command, it's often easiest to animate your screens using AMAL instead as this generates a much smoother effect.

CHANNEL n TO SCREEN OFFSET d

This assigns AMAL program number *n* to a screen *d*, for the purpose of hardware scrolling. The X and Y registers now refer to the section of the screen which is to be displayed through your TV. Changing these registers will scroll the visible screen area around the display. Here's an example:

```
Screen Open 0,320,500,32,lowres : Rem Open an extra tall screen
Screen Display 0,,45,320,250
Load Iff "AMOS_DATA:IFF/Magic_Screen.IFF"
Screen Copy 0,0,0,320,250 To 0,0,251
```

```
Screen 0 : Flash off : Get palette (0)
Channel 0 to Screen Offset 0
Amal 0,"Loop: Let X=XM-128; Let Y=YM-45; Pause; Jump Loop"
Amal On : Wait Key
```

This program allows you to scroll through the screen using the mouse. Try moving the mouse in direct mode. For a further example of hardware scrolling, see **Example 14.5**.

Changing the screen size

CHANNEL n TO SCREEN SIZE s.

This allows you to change the size of a screen using AMAL. *s* is the number of the screen to be manipulated. Registers X and Y now control the width and height of your screen respectively. They're similar to the W and H parameters used by the SCREEN DISPLAY command. Example:

```
Load Iff "AMOS_DATA:IFF/Magic_Screen.IFF",0
Channel 0 To Screen Size 0
Screen Display 0,,,320,1 : Rem Set the screen size to 1
A$="Loop: For R0=0 To 255 ; Let Y=R0 ; Next R0; "
A$=A$+"For R0=0 To 254; Let Y=255-R0; Next R0; J Loop"
Amal 0,A$ : Amal On : Direct
```

Rainbows

CHANNEL n TO RAINBOW r

This option generates a rainbow effect within an AMAL program. As usual *n* is the number of an animation channel from 0 to 63. *r* is an identification number of your rainbow (0-3).

X holds the current BASE of your rainbow. This is the first colour of your rainbow palette to be displayed. Changing it will make the rainbow appear to turn. Y contains the line on the screen at which the rainbow effect will start. If you alter this value, the rainbow effect will move up or down. All coordinates are measured in **hardware** format.

Register A stores the height of your rainbow on the screen. A demonstration of this system can be found in **11.** See the AMOS Basic RAINBOW command for more details.

Advanced Techniques

The AUTOTEST system

Normally all AMAL programs are performed in strict order from start to finish. Inevitably some commands such as Move and For...Next will take several seconds to complete. Although this will be fine in the vast majority of cases it may lead to significant delays in the running of certain programs. Take the following simple program:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
```

```
Sprite 8,130,50,1
Amal 8,"Loop: Let R0=XM-X; Let R1=YM-Y; Move R0,R1,50; Jump Loop"
Amal On : Direct
```

As you move the mouse, the sprite is supposed to follow it around on the screen. However in practice the response time is quite sluggish, because the new values of XM and YM are only entered after the sprite movement has totally finished. Try moving the mouse in a circle. The octopus is completely fooled!

Autotest solves this problem by performing your tests at the start of every VBL, before continuing with the current program. Your tests now occur at regular 1/50 intervals, leading to a practically instantaneous response!

Autotest commands

The syntax of Autotest is:

AUtotest (tests)

tests can consist of any of the following AMAL commands.

Let

L reg=exp

This is the standard AMAL Let instruction. It assigns the result of an expression to register *reg.*

Jump

Jump label

The Jump command jumps to another part of the current autotest. *Label* is defined using the colon ":" and **must** lie inside the autotest brackets.

e**X**it

Leaves the autotest and re-enters the main program from the point it left off.

Wait

Wait turns off the main AMAL program completely, and only executes the Autotest.

If

In order to simplify the testing process inside an autotest routine there's a specially extended version of the AMAL If statement. This allows you to perform one of three actions depending on the result of the logical expression *exp.*

if exp Jump L (Jumps to another part of the autotest)
If exp Direct L (Chooses part of the program to be executed after an autotest)
If exp eXit (Leaves autotest)

On

ON restarts the main program again after a previous Wait instruction. This lets you wait for a specific event such as a mouse click without wasting valuable processor time.

Direct

Direct label

Direct changes the point at which the main program will be resumed after your test. AMAL will now jump to this point automatically at the next vertical blank period. Note that *label* **must** be defined outside the Autotest brackets.

Inside Autotest

Here's the previous example rewritten using the Autotest feature.

```
Load "AMOS_DATA:Sprites/octopus.abk"
Sprite 8,130,50,1 : Get Sprite Palette
A$="AUtotest (If R0<>XM Jump Update"
A$=A$+"If R1<>YM Jump Update else eXit"
A$=A$+"Update: Let R0=XM; Let R1=YM; Direct M)" : Rem End of autotest
A$=A$+"M: Move R0-X,R1-Y,20 Wait;" : Rem Try changing 20 to different values!
Amal 8,A$ : Amal on
```

The sprite now smoothly follows your mouse, no matter how fast you move it. The action of this program is as follows:

Every 50th of a second the mouse coordinates are tested using the XM and YM functions. If they are unchanged since the last test, the Autotest is aborted using the eXit command. The main program now resumes precisely where it left off.

However if the mouse has been moved, the autotest routine will restart the main program again from the beginning (label **M**) using the new coordinates in XM and YM respectively.

Timing considerations

UPDATE EVERY *(Save some time for your Basic programs)*

UPDATE EVERY n

Although most AMAL programs are performed practically instantaneously, any objects they manipulate need to be explicitly drawn on the Amiga's screen.

The amount of time required for this updating procedure is unpredictable and can vary during the course or your program. This can lead to an annoying jitter in the movement

patterns of certain objects.

The UPDATE EVERY command slows down the updating process so that even the largest object can be redrawn during a single screen update. This regulates the animation system and generates delightfully smooth movement effects.

n is the number of vertical blank periods (50ths of second) between each screen update. In practice you should start off with a value of two, and gradually increase it until movement is smooth.

One useful side effect of UPDATE EVERY, is to reserve more time for Basic to execute your programs. With judicious use of this instruction, it's sometimes possible to speed up your programs by as much as 30%, without destroying the smoothness of your animation sequences.

Beating the 16 object limit

SYNCHRO *(Execute an AMAL program directly)*

SYNCHRO [ON/OFF]

Normally AMOS Basic will allow you to execute up to 16 different AMAL programs at a time. This limit is determined by the overall speed of the Amiga's hardware. Each AMAL program takes its own slice of the available processor time. So if you're using the standard interrupt system, there's only enough time to execute around 16 separate programs.

The SYNCHRO command allows you to exceed this restriction by executing your AMAL programs directly from Basic. Instead of using interrupts, all AMAL programs are now run using a single call to the SYNCHRO command. Since AMAL programs execute far faster than the equivalent Basic routines, your animations will still be delightfully smooth. But you will now be able to decide when and where your AMAL routines will be performed in your program.

One additional bonus is that you can now include collision detection commands such as Bob Col or Sprite Col directly in your AMAL routines. These are not available from the interrupt system as they make use of the Amiga's blitter chip. This would be impossible using interrupts.

Before calling SYNCHRO you first need to turn off the interrupts with SYNCHRO OFF. It's important to do this **before** defining your AMAL programs, otherwise you won't be allowed to use channel numbers greater than 15 without an error.

Due of the sheer power of the animation system, it's nearly possible to write entire arcade games completely in AMAL. This leaves your Basic program with simple jobs such as managing the hi-score table and loading your attack waves from the disc. The results will be indistinguishable from pure machine code. A good example is Cartoon Capers, the first commercial games release that's written entirely in AMOS.

A demonstration of SYNCHRO can be found in **Example 14.6** from the MANUAL folder.

STOS compatible animation commands

The original STOS Basic included a powerful animation system which allowed you to move your sprites in quite complex patterns using interrupts. At the time, these commands were hailed as a breakthrough.

Although they've now been overshadowed by the AMAL system, they do provide a simple introduction to animation on the Amiga. So AMOS provides you with the entire STOS animation system as an extra bonus!

If you're intending to convert STOS programs to AMOS, you'll need to note the following points:

- Unlike STOS, the movement patterns in AMOS Basic can be assigned to any animation channel you like. The Move commands can therefore be used to move bobs, sprites or screens, using exactly the same techniques.

  As a default, all animation channels are assigned to the equivalent hardware sprites. In practice you may find it easier to substitute blitter objects as these are much closer to the standard STOS Basic sprites. Add a sequence of CHANNEL commands to the start of your program like so:

```
Channel 1 to bob 1
Channel 2 to bob 2
:    :    :
```

  Don't forget to call DOUBLE BUFFER during your initialisation procedure, otherwise your bobs will flicker annoyingly when they are moved.

- The same channel can be used for both STOS animations and AMAL programs. So it's easy to extend your programs once they've been successfully converted into AMOS Basic. The order of execution is:

```
AMAL
MOVE X
MOVE Y
ANIM
```

MOVE X *(Move a sprite horizontally)*

MOVE X n,m$

MOVE X defines a list of horizontal movements which will be subsequently performed on animation channel number *n*.

n can range from 0 to 15 and refers to an object you have previously assigned using the CHANNEL command. *m$* contains a sequence of instructions which together determine both the speed and direction of your object. These commands are enclosed between brackets and are entered using the following format:

```
(speed,step,count)
```

There's no limit to the number commands you can include in a single movement string, other than the amount of available memory.

speed sets a delay in 50ths of a second between each successive movement step. The speed can vary from 1 (very fast) to 32767 (incredibly slow).

step specifies the number of pixels the object will be moved during each operation.

If the step is positive the sprite will move to the right, and if it is negative it will move left.

The apparent speed of the object depends on a combination of the speed and step size. Large displacements coupled with a moderate speed will move the object quickly but jerkily across the screen. Similarly a small step size combined with a high speed will also move the object rapidly, but the motion will be much smoother. The fastest speeds can be obtained with a displacements of about 10 (or -10).

count determines the number of times the movement will be repeated. Possible values range from 0 to 32767. A *count* of 0 performs the movement pattern indefinitely.

In addition to the above commands, you can also add one of the following directives at the end of your movement string.

The most important of these extensions is the L instruction (for loop), which jumps back to the start of the string and reruns the entire sequence again from the beginning. Example:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Sprite 1,130,100,1 : Rem Define Sprite 5
Move X 1,"(1,5,60)(1,-5,60)L"
Move On
```

The E option allows you to stop your object when it reaches a specific point on the screen. Change the second to last line in the above example to:

```
Move X 1,"(1,5,30)E100"
```

Note that these end-points will only work if the x coordinate of the object exactly reaches the value you originally designated in the instruction. If this increment is badly chosen the object will leap past the end-point in a single bound, and the test will fail. Example:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Channel 1 To Sprite 8 : Channel 2 To Sprite 10
Print At(0,5)+"Looping OK"
Sprite 8,130,100,1
Move X 1,"(1,10,30)(1,-10,30)L"
Move On
Print At(0,10)+"Now press a key" : Wait Key
Sprite 10,140,150,2
Move X 2,"(1,15,20)L" : Move On 2
Print At(0,15)+"Oh dear!" :Wait Key
```

MOVE Y *(Move an object vertically)*

MOVE Y n,m$

This instruction complements the MOVE X command by enabling you to move an object vertically along the screen. As before, *n* refers to the number of an animation sequence you've allocated using the CHANNEL command, and ranges between 0 and 15.

m$ holds a movement string in an identical format to MOVE X. Positive displacements now correspond to a downward motion, and negative values result in an upward

movement. Examples:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Channel 1 To Sprite 8 : Sprite 8,130,10,1 : Rem Install sprite
Move Y 1,"10(1,1,180)L" : Rem Loop sprite from 10,10 to 190,10 continually
Channel 2 To Screen Display 0 : Rem Assign the screen postion to channel 2
Move Y 2,"(1,4,25)(1,-4,25)" : Rem Moves screen up and down
Move On : Wait Key
```

MOVE ON/OFF *(Start/stop movements)*

MOVE ON/OFF [n]

Before your movement patterns will be executed they need to be activated using the MOVE ON command.

n refers to the animation sequence you wish to start, and can range from 0 to 15. If it is omitted then all your movements will be activated simultaneously.

MOVE OFF has exactly the opposite effect: It stops the relevant movement sequences in their tracks.

MOVE FREEZE *(Temporarily suspend sprite movements)*

MOVE FREEZE [n]

The MOVE FREEZE command temporarily halts the movements of one or more objects on the screen. These objects can be restarted again using MOVE ON.

n is completely optional and specifies the number of a single object to be suspended by this instruction.

=MOVON *(Return movement status)*

x=MOVON(n)

MOVON checks whether a particular object is currently being moved by the MOVE X and MOVE Y instructions. It returns -1 (true) if object *n* is in motion, and 0 (false), if it is stationary. Do not confuse this with the MOVE ON command.

Note that MOVON only searches for movement patterns generated using the MOVE commands. It will not detect any animations generated by AMAL.

ANIM *(Animate an object)*

ANIM n,a$

Anim automatically flicks an object through a sequence of images creating a smooth animation effect on the screen. These animations are performed 50 times a second using interrupts, so they can be executed simultaneously with your Basic programs.

n is the number of the channel which specifies a sprite or bob to be animated by this

instruction.

a$ contains a series of instructions which define your animation sequence. Each operation is split into two separate components enclosed between round brackets.

image is number of the image to be displayed during each frame of the animation. *delay* specifies the length of time this image will be held on the screen (in 50ths of a second). Example:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Channel 1 To Sprite 8 : Sprite 8, 200,100,1
Anim 1,"(1,10)(2,10)(3,10)(4,10)"
Anim on : Wait key
```

Just as with the MOVE instruction, there's also an L directive which enables you to repeat your animations continuously. So just change the ANIM command in the previous example to the following:

```
Anim 1,"(1,10)(2,10)(3,10(4,10)L"
```

ANIM ON/OFF *(Start an animation)*

ANIM ON/OFF[n]

ANIM ON activates a series of animations which have been previously created using the ANIM command. *n* specifies the number of an individual animation sequence to be initialised. If it is omitted then all current animation sequences will be started immediately.

ANIM OFF [n]

This halts one or more animation sequences started by ANIM ON.

ANIM FREEZE *(Freeze an animation)*

ANIM FREEZE [n]

ANIM FREEZE temporarily freezes the current animation sequence on the screen. *n* chooses a single animation sequence to be suspended. If it's not included, all current animations will be affected. They can be restarted at any time with a simple call to the ANIM ON instruction.