Above are four members of Bullfrog, the people behind this tutorial. Ever heard of *Populous* 1 and 2? These are the people responsible. Bullfrog don't go in for really complicated, in-depth programming techniques – they like to keep things simple.

Your turn!

How to program your own games in assembler

If you've ever wanted to become a games programmer, now's your chance. A complete assembly language on the coverdisk, plus a 'beginners' kit' of code and a complete tutorial from Bullfrog coder Scott Johnston. Don't think you can learn from a magazine article? Believe it – that's how Scott learnt!

What better way to learn to program than from experts like Bullfrog, the programming team behind the legendary *Populous* 2? And, what better person to learn from than a coder who now works on the Bullfrog team, and yet learnt how it's done from a magazine article?

Bullfrog leader Peter Molyneux explains: "A year or so, when *ST Format* (*Amiga Format*'s sister magazine for the Atari ST 16-bit computer) ran the series we wrote for them teaching people how to program, Scott had applied for a job as a trainee with us."

Bullfrog regularly takes on trainees and teaches them how to code, as well as taking youngsters on work-experience placements, which is why they have a well-established training system that forms the basis for this series. Anyway, back to Peter...

"We didn't have a job for Scottt to do at the time, so we couldn't take him on. Anyway, Scottt went away to college and taught himself how to program from the series in *ST Format*. A year later we were able to take him on, and by then he already knew how to code." Which only goes to show that you can do it too.

What we've got for you is the assembly language *DevPac 2*. Assembly language is used in nearly all professional Amiga games, because nothing else is quick enough. We've also got a code 'shell' provided by Bullfrog, which is exactly as used in *Populous* 2 and handles a lot of the tricky setting up of the

Continued overleaf

hardware, leaving you to get down to the nitty-gritty of creating a game.

During the course of the series, the code provided by Bullfrog will build up into a complete arcade game. It might not be the most stunning thing you've ever seen, but it will be all your own work!

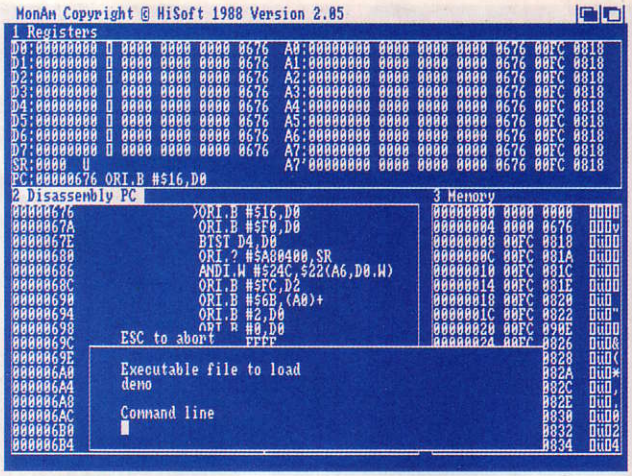
Just before we start, let's take a look at the basic idea of how a game works. The whole thing is built around what's known as the 'main game loop'. This is a process of checking, every fraction of a second, everything that repeats. A typical main game loop for an arcade game might be something like this...

One: check the joystick. Two: move the sprite accordingly. Three: check for collisions; if collision has taken place, deplete energy. Four: check game status counts; if energy is gone, it's game over. Then it goes back to the start of the loop.

This would give you a game in which a sprite moves around bumping into things and that eventually dies. Obviously, you then have to add in other things: if the fire button is pressed, start off a bullet animation; if enemies are moving around the screen, move them after the sprite; check if bullets have hit enemies, and if so get rid of them and add to the player's score; and so on. That's the basis.

Where we have to start off, of course, is getting a sprite on the screen and moving it around. Which is what we'll cover this month. But first, let's go over to Scott to introduce a few basics...

Any serious programmer will tell you that the beauty of programming in assembly language is the sheer speed you can achieve. It's easy making your Amiga produce quality animation, but when you want to turn an animation sequence into a game you need to start reading the position of the joystick, responding to the player's movements and keeping track of the enemies. To try to cram all these routines into a Basic program and still keep it running up to speed would be a nightmare.



This is MonAm2, the built-in Devpac monitor. You won't need to use it for this issue, but for the curious, it's a program for taking raw machine code and turning it back into assembler. It's also rather useful for finding errors.

Assembly language is so fast it's the ideal alternative. Currently the most popular assembly language programming tool is HiSoft's *Devpac*, although they have brought out the follow-up package, *Devpac 2*, which provides a comprehensive editor and assembler enabling you to create and edit your own programs. How you edit, assemble and then run a program is explained fully in the separate instructions box.

Assembly language programs correspond to your Amiga's machine language, so they are far more complex to write than Basic programs, but don't let that put you off.

This issue we begin a full tutorial on programming in assembly language and will keep it so simple, even the cat could use it. To start off, we're going to look at how you go about constructing an animation sequence.

Included on the disk is a simple 'shell' for writing programs in assembler, plus routines used to create *Populous* 2, including some of the sprite draws and all of the screen draws. It sets you up with all you need to swap screens, draw sprites, and so on.

With this shell you don't have to wade through loads of reference manuals to find out how to set up screens or display sprites: it's all been done for you. Over the course of several months we will be putting together a very simple game, but there are several things you will need so you can stick it through to the end:

- Enthusiasm. Yep, I know, it sounds obvious, but you will always be just a standard programmer unless you really want to do it well.

- It would be nice if you read a 68000 assembler book. (*Amiga Format* recommends Abacus's *Amiga Machine Language* – ISBN 1-55755-025-5 – available from Computer Manuals 021-706 6000).

- Do you know what binary and hexadecimal are? If not then try and find out. Get a reference manual out of the library.

Right, enthusiasm proven, we're ready to go on. As with any programming project whatever language you are using, the first stage is to design the sprites you need by using an art package. You don't need to do this yourself because we've included sprites on the disk, but here's how it's done, and next month we'll

HOW TO USE THE DISK

We've tried to make using *Devpac* as painless as possible. When you boot the disk, you'll find yourself in the text editor of the program. This is where you write your programs, although for this first part the vast majority of the programs have already been written for you.

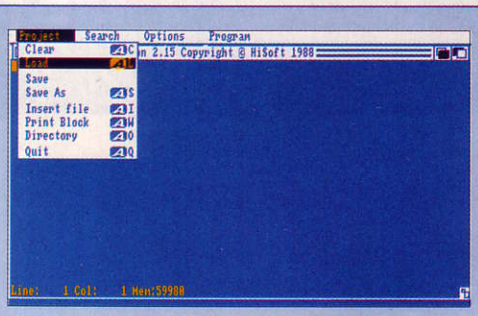
The most important thing to remember about machine code is that the code itself is all numbers. Computers only understand numbers, or object code as it's sometimes called. People, however, don't write programs using just numbers, but instead use a special language called assembly language. This can't be understood directly by the computer and is called source code.

An assembler like *Devpac* will take all the source (assembly) code and translate it directly into object (machine) code. There's lots more it can do too. For instance, there's a monitor for viewing your programs as they are working, but that is the bare bones of *Devpac*. Incidentally, if you accidentally get into the monitor, press Return a couple of times, and then Ctrl and C to return to the editor.

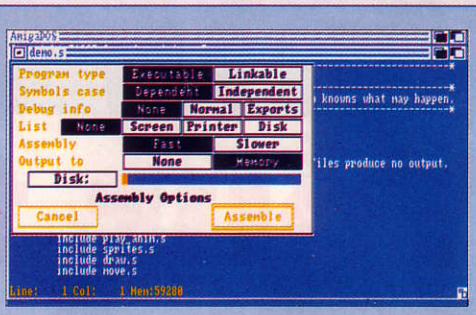
Finally, a couple of notes about special *Devpac* commands. If a line begins with a semicolon (;) then *Devpac* will ignore it. Useful for hiding parts of a program you don't need.

When you see a command like "DC.B" or "DC.W" or "DC.L" that's just a data area for variables to be stored in.The command word "Even" is for making sure data areas are even in length (if not, the Amiga tends to crash).

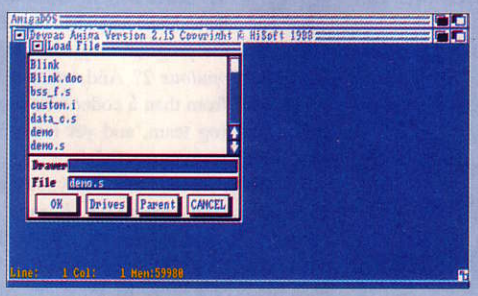
Anyway, here's how to load source code:



1 Move the mouse pointer to the top left of the screen and hold down the right mouse button. A menu will drop down. Select 'Load' from the Project menu, and a file requester will appear.



3 To turn this into machine code, select 'Assemble' from the Program menu. A list of options will appear. Just click on Memory, and then on Assemble. *Devpac* will set the assembler going and ask you to press a key when it has finished.



2 A handy tip for this file requester is to click on the slider on the right-hand side and move it up and down. The list of files will be automatically sorted alphabetically. Find and double click on the file called Demo.s. It will load and display.

4 To set the program going, just select the 'Run' option from the Program menu. The demo will run, displaying a sprite – type Q to quit.

provide a utility that will let you use your own sprites in the game we're creating.

There are several common art packages which are suitable, but Bullfrog tend to process their pre-drawn visuals with *Deluxe Paint III* or *IV*. In fact, most programmers who work on an Amiga use *DPaint*. But whatever art package you are using, the first step is to set up a grid of eight by eight pixels.

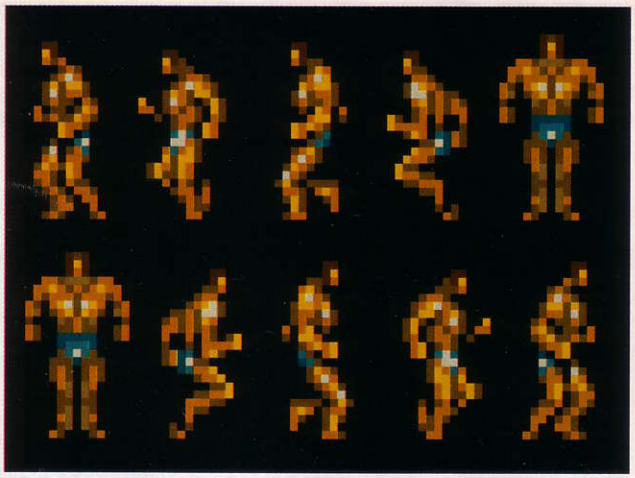
Make a checkerboard or grid in which to draw the sprites. "Personally, I prefer the checkerboard approach, but I've seen graphic artists fight about which is the best method," says Bullfrog boss Peter Molyneux. Now go ahead and design your sprites keeping them within the boundaries of your grid.

Once your sprites for the frames of animation are drawn, you must place them in order starting at the top-left point of the screen and working your way along. Once you reach the right-hand side of the screen, move down a line and continue.

It's vital at this point to understand the Amiga's screen format. Each pixel in a low-resolution screen is stored as four bits in four different four-bit groups called words. These are known as bit planes, and because there are four you have 16 possible combinations (1 bit times four planes, or four to the power of two) for each pixel – hence a palette of 16 colours.

The four different words are stored at different points of memory, with all the words of the first plane (Plane 0) stored in memory together, followed by plane 1 and so on. The memory looks something like this:

```
Start of Plane 0
8000 bytes
Start of Plane 1
8000 bytes etc..
```



A total of nine sprites are used for the man sprite in the demo. They probably look distorted and unreal to most people at this scale, but they look just fine on screen.

The most efficient method of storing a sprite in memory, so you can access it for quick drawing, is to store each line of the sprite as four words for the first 16 pixels of the line, four words for the next 16 pixels and so on. This is why almost all sprite routines cater only for sprites which are a multiple of sixteen pixels wide – 16, 32, 48, 64 and so on.

As we have said, included on next month's disk will be a convert program which will accept IFF files from art packages like *DPaint*. It will then convert your graphics into data usable by the program. Just for this month, stick with our graphics.

OK, that's the boring introduction done, let's have a look at the disk. If you first assemble Demo.s and then run Demo (instructions on assembling and running are in the separate box), all that should happen is that we display a sprite on the screen. To exit from the program press the Q Key. Great. There you go. What? You want more?

We will never be changing demo.s – well, at least not very often. Whenever you want to assemble this program, first save out the file you were using and then load in Demo.s. When assembling this, switch the OUTPUT TO MEMORY and then you can test the program just by using the RUN command. This way, you won't be saving the changed demo on to the disk over the old one.

Now let's try and move the sprite. Have a look in draw.s in the routine:

```
_draw_all
```

and you will see we have got the following bit of code:

```
move.w  #0,d0
move.w  #0,d1
move.w  #0,d2
```

Now, if we change these to...

```
move.w man_x,d0
move.w man_y,d1
move.w man_frame,d2
```

Right, now the impatient ones of you will have assembled that. No change is visible, of course not, but if you go into move.s and remove the indicated ;'s our sprite should start to move to the right and return to the start when he reaches the end of the screen.

The way these instructions work is as fol-

Continued overleaf

THE HEART OF THE MACHINE: THE MC68000 CENTRAL PROCESSOR

The chip that actually makes the decisions and runs programs in most Amigas is the Motorola Corporation 68000. Some Amigas have more powerful versions of this – big brothers, if you like – but the differences are fairly minor, save that bigger machines go faster.

If you're unsure of how computer memory works, all I can do is advise you to read and digest Answerfile, both on p221 of this issue and on p199 of last month's *Amiga Format*. That will give you a lot of insights into the way that the Amiga handles information.

The way the 68000 works is not too tricky to grasp. Inside the chip are stores for numbers, a bit like memory, save that they can be manipulated directly by the chip. They're called registers and can be added, multiplied and tested for. If a number gets to big or small, a jump can be made to a different part of a program – this is how the 68000 makes decisions.

There are different types of registers. The ones that point to areas in the Amiga's memory are called address registers. There are eight of them and their names are fairly dull: A0, A1, A2, A3, A4, A5, A6 and A7. Using the Bullfrog's routines on the Collection Coverdisk, you will rarely need to use the address registers.

Far more important for our purposes are the data registers. These are used to store numbers to be manipulated. They're called D0, D1, D2, D3, D4, D5, D6 and D7. There are other data registers, but leave them alone for the minute. In this issue, you'll only need to bother with the first three – these are used to store the X coordinate (D0), the Y coordinate (D1) and the frame number of the sprite (D2).

There are other registers, but the most important of these are the PC (Program Counter) and Status registers (usually called the SR). The PC keeps the address of where the program is currently up to in memory – when you jump to a different address, all that happens is the PC is given a new place to start.

The Status register holds many flags, which keep track of things like whether or not the last instruction resulted in zero, or a negative number, and similar conditions.

So in this example, when the animation frame number is compared to 4 and is less than or equal to that number, the program continues. When it is greater than that, the frame counter is set back to zero. There are other frames of animation, but if you use them then your man will turn towards you while walking.

There are other registers for doing different tasks inside the MC68000, but you don't need to know about them yet. What is more important is that you appreciate the above – that's all you need to understand to use this tutorial.

For the curious, I'll tell you about some of the other bits of machine code – the parts that Bullfrog have steered you around by including lots of routines to do the dirty work for you.

There is a block of memory which is reserved for use by the custom chips of the Amiga. These are accessed just like normal memory, but you can't use them for storing information. They come in three basic types; read-only registers, which the 68000 can look at but not talk to; write-only registers, which the 68000 can talk to but not read; and read/write registers, which it can read and talk to.

These hardware registers are the gateway between the 68000 and the Amiga. For instance, if you set one of the binary switches in a certain register, the display will change from low resolution to high resolution.

It's not all that simple, not by a long shot. For instance, to make the Paula chip play a sound sample, you have to tell it where the sample is in memory, how fast to play and which channels to use. Bear in mind that you have to do this using hexadecimal numbers and you see that talking to the hardware directly can be a nightmare.

If you want to see just how involved it is, use the Devpac editor to load in the other files that are listed in demo.s. Some of them are fairly large. The amount of programming you have to do for even the smallest tasks – like opening a custom screen, reading the joystick and putting up a sprite – is quite enormous.

However, for those interested, the Bullfrog's routines will prove very useful. Many people are stumped with programming because they don't have enough examples to work from.

The trouble is that up to now there hasn't been much available in the sense of professional machine code for you to look at and learn from. Try to learn as much as you can from reading the source – most of it has documentation.

lows. First we increment (increase) the value of the man's x co-ordinate by 1 every turn. The value is then compared with 304 (the edge of the screen) and if it is greater than this the position is reset to 0. Otherwise we jump over the reset command. The value is then stored back into man_x ready for the next draw.

You can change the value in the add command to increase the speed of movement.

Taking a moment to explain a couple of things: d0 is a register, which is basically just an area of RAM in which you store a value. A variable is a number that changes, basically, so you can add to a variable to affect what your program is doing. Hope that's simple enough to be going on with. Anyway, the way all these things work will become clear as you practice.

A small challenge for you: see if you can write the code to move the man down the screen. The code will be basically the same as the 'move x' stuff, but remember the screen size is 320 by 200 pixels, and that the label 'lessthanx' has already been used, so use something different.

A note on labels: try and keep them meaningful or you will forget what they are doing later on. Having six labels with the name '.here' is not very helpful.

We should, by now, have got the man moving down and across the screen. But this points out a problem: he's moving as if he is floating. The man isn't actually doing any walking. Let's put a bit of animation into the man.

To do this we take the same routine as was used in the movement, but change a few things. If you're still unsure of moving the man down the screen, the next bit of code should sort you out – it will animate the sprite, and should be added into Move.s, before the final RTS command.

```
move.w man_frame,d2
add.w #1,d2
cmp.w #4,d2
ble.s  .lessthana
move.w #0,d2
.lessthana
move.w d2,man_frame
```

Be careful, we only have four frames of animation to play with, so don't set the compare too high.

Compile this and have a look. He is changing frames just a little bit fast – too fast, in fact – so we need to slow him down. To do this change the line

```
cmp #4,d2
```

to

```
cmp #4*4,d2
```

Also, scale the number down. After the line

```
move.w man_frame,d2
```

in draw.s, place the following piece of code:

```
asr.w  #2,d2
```

Assemble this, and the man should be walking a bit slower.

So, in just one month we've got an animated sprite moving around the screen and learnt some of the most important commands. Next month, we'll need to put our man under joystick control. See you then.

COMMAND OVERVIEW

```
move.w #1,d0
```

The move command is one of the most heavily used commands. For simplicity we are using move.w – the w means word-sized data only. It is the equivalent of the basic LET statement. It loads x (a number) into d0.

```
add.w #1,d0
```

The add command does just that: it adds the value of 1 to d0.

```
sub.w #1,d0
```

The sub command is similar to the add but subtracts instead.

```
cmp.w #1,d0
```

The compare command compares the value of 1 with the value in d0. When used in conjunction with a branch it's similar to the IF statement you'd find in the Basic language.

Branches

The branch checks on the flags set by the compare statement. There are 15 different versions of this command:

```
bra   Branch always. Same as GOTO
bcc   Branch if carry clear
bcs   Branch if carry set
beq   Branch if equal
bge   Branch if greater than or equal to
bgt   Branch if greater than
bhi   Branch if higher
ble   Branch if less than or equal to
bls   Branch if low or same
blt   Branch if less than
bmi   Branch if negative
bne   Branch if not equal to
bpl   Branch if positive
bvc   Branch if overflow clear
bvs   Branch if overflow set
```

Don't worry if you don't understand what all of these do. By using them you'll soon get the hang of their particular functions. Anyway, when the condition is true the branch is taken. If not then we move on to the next instruction.

Labels

You will notice that some of the labels have a '.' in front of them: these are called local labels. That means that you can use words like .finished more than once, though not inside the same routine. Routines are defined by labels which have not got a period '.' in front of them and are finished with an RTS command.

```
jsr
```

Jump to subroutine. This command is basically the same as the GOSUB command.

```
rts
```

This is basically the same as the RETURN command.

```
asr.w #1,d0
```

The asr command means Arithmetic Shift Right and works by moving a binary number to the right a definite number of times.

For example, on a

```
asr.w #1,10
```

we get an answer of 5. This is because 10 in binary looks like this:

```
1010  =  10
```

and when we move it to the right by one it becomes:

```
0101  =  5
```

There are many more commands for shifting, rotating, and slicing, but there isn't really room to fit them in here.

If you want the best, most authoritative and comprehensive guide to the 68000, try to get a copy of Sybex's *Programming the 68000* (£23.95 from Computer Manuals 021-706 6000). It's very handy for looking up particular commands.

OOOPS!

If you quit out of *Devpac*, you'll get to some readme files on the disk. Unfortunately, looking at them can't be done on Workbench 2 machines. Worse, it causes other Amigas to crash! The reason is that, because of an oversight, the wrong version of the text displaying program (ppmore) has been included.

Putting the right version on to the disk (copied from the *Pinball/Caesar* Coverdisk) is fairly easy. Boot up with Workbench, double click on the Shell and type:

```
Copy from Coverdisk39b:c/ppmore to Devpac2:c
```

After a disk swap or two, the right program will be copied over and you can view the documents – they're not at all necessary when using *Devpac*, but we thought we'd tell you about it just to be on the safe side.

UPGRADING TO THE LATEST *DEVPAC* – AND NEWS OF A GREAT PROGRAMMING COMPO!

When you've been using *DevPac* for a bit, you might decide it's exactly what you've been looking for in a programming package. You might just be bitten by the bug! If so, you can always upgrade to the highly-specified, updated version of the top assembly language.

***DevPac* 3 is probably the only ever Format Gold award-winning programming language. To coincide with this article, HiSoft are arranging a special upgrade offer at a great price: contact them on 0525 718181 for details.**

And while we're about it, here's news of a great competition we'll be running in a few issues' time. Next issue, DMI Games will be supplying their *PP Hammer* sprites and we'll show you how to load them into your programs. In three months' time, DMI will be offering some amazing prizes to the best program created using *PP Hammer*. And, what's more, if it's good enough that program will be published! But that's just a taste of what we've got for the future: see you next month for more programming!