Bare-Metal Amiga Programming

For OCS, ECS and AGA

E. Th. van den Oosterkamp

ISBN: 9798561103261

Web site: www.edsa.uk/amiga

Author: edwin@edsa.uk

Corrections

Page 36, description of the AUDxVOL registers

Omitted was the fact that setting of bit 7 will always yield maximum volume, regardless of the state of bit 0 to 6.

Page 85, description of the DWSTRT and DWSTOP registers

Improve grammar and remove reference to calculation of the start position. The start position is not calculated, but rather chosen by the programmer.

Page 107, Independently moving bitplanes example

To calculate the new X and Y positions the new relative value and the old relative value were subtracted. In the new version they are added instead. For the working of the example it does not make much difference, but now the mouse to X/Y coordinate routine no longer swaps up with down and left with right.

Page 242, Starting and stopping safely

The routine uses a WORD in the Exec library to check if the CPU is an 68000 or an 68010 or higher. Unfortunately the BTST instruction tested the wrong byte of the word, and therefore failed to detect the correct CPU. This in turn meant that the VBR location was never correctly stored.

Page 244, Starting and stopping safely

When its DMA is enabled the Copper will continue to run from where it was stopped. This meant that the Copper would only use the restored system coplist after the next vertical blank. Writing to the COPJMP1 register before restarting DMA now ensures that the Copper will run the restored system coplist immediately.

The following pages in this document are the corrected pages of the book, where the corrections are marked in purple.

Bit	Name	Function
15-8		Unused, set to zero
7	VOL7	Set to 1 for max volume, set to 0 to use the value of VOL(0-6).
0-6	VOL(0-6)	Channel volume

Each audio channel has its own volume register. When bit 7 is clear the channel’s volume is defined by the value of bits 0 to 6, where a value of 0 will mute the channel. When bit 7 is set the channel’s volume will be set to maximal, regardless of the setting of the other bits. At maximum volume the voltage on the Amiga's audio connectors is between -0.4 and 0.4 Volt.

AUDxDAT - DAC data register

This register is used by the DMA engine to transfer two audio samples from memory to the DAC. When audio DMA is used (automatic mode) there is no need for the CPU to access this register. When instead of DMA the CPU is used to play back audio samples (direct mode) then the CPU needs to copy the next two samples for the DAC into the AUDxDAT register. The audio hardware can be setup to generate an interrupt just before the new samples are required.

DMACON - DMA control

Below are shown the bits of the DMACON register that are related to the audio DMA, for the full register please consult the “Direct memory access” chapter.

DMACON	DMA Control	W $096
Bit	**Name**	**Function**
15	SET/CLR	Set to 1 to set bits, set to 0 to clear bits
9	DMAEN	Set to 1 to enable DMA, set to 0 to disable all DMA
3	AUD3EN	Audio channel 3 DMA enable
2	AUD2EN	Audio channel 2 DMA enable
1	AUD1EN	Audio channel 1 DMA enable
0	AUD0EN	Audio channel 0 DMA enable

Enabling the DMA for a particular audio channel will directly put that channel into automatic mode. When the DMA for an audio channel is disabled then that audio channel will from then on again be operating in direct mode.

The range of these positions can be increased on ECS and AGA by writing to the DIWHIGH register after writing to DIWSTRT. Originally the Hx bits were named Hx(7-0), but these were renamed with the introduction of the AGA chipset to Hx(9-2) due to the addition of bits named H0 and H1 in the DIWHIGH register.

DIWSTOP - Display window stop position (bottom right corner)

DIWSTOP		Display window stop position W $090
Bit	**Name**	**Function**
15-8	Vx(7-0)	Vertical stop position of the display window
7-0	Hx(9-2)	Horizontal stop position of the display window

For the calculation of the stop position please refer to the section titled “The display window”. The range of the two stop positions can be increased on ECS and AGA by writing to the DIWHIGH register after writing to DIWSTOP. Similar to the Hx bits in the DIWSTRT register the bits in the DIWSTOP register were renamed from Hx(7-0) to Hx(9-2) with the introduction of AGA.

DIWHIGH - Display window position (ECS/AGA)

DIWHIGH		Display window position (ECS/AGA) W $1E4
Bit	**Name**	**Function**
15-14		Unused set to 0
13	H10	Most significant horizontal stop bit
12-11	H0-H1	ECS: Unused, set to 0. AGA: Horizontal least significant stop bits
10-8	Vx(10-8)	Most significant vertical stop bits
7-6		Unused set to 0
5	H10	Most significant horizontal start bit
4-3	H0-H1	ECS: Unused, set to 0. AGA: Horizontal least significant start bits
2-0	Vx(10-8)	Most significant vertical start bits

This register was added to ECS in order to increase the range of the start/stop locations for the display window. For AGA four more bits were added that increase the resolution of the horizontal start and stop locations. Writing to DIWSTRT or DIWSTOP will reset the contents of DIWHIGH in order to keep compatibility with OCS systems. To use the functionality provided by DIWHIGH on ECS/AGA systems it is crucial that DIWHIGH is always written to after writing to DIWSTRT and DIWSTOP.

The example uses scrolling in both directions and just like the horizontal scrolling example it uses a DMA fetch start that is one DMA period earlier than the display window start. To account for this additional fetch of 2 bytes and the fact that the playfield is 40 bytes larger than the display window the BPLxMOD registers are set to 38:

```
MOVE.W   #40-2,BPL1MOD(a5)      ; Odd bitplane 40 bytes larger than playfield
MOVE.W   #40-2,BPL2MOD(a5)      ; Even bitplane 40 bytes larger than playfield
```

The rest of the setup is similar to the setup of the horizontal scrolling example and this example will also use the vertical blank interrupt for synchronisation.

Moving the mouse will increment or decrement two 8 bit counters, depending on the direction the mouse is moved in. The relative change in mouse position can then be calculated from the changes of the two counters. For the first calculation to make sense the start position is required:

```
MOVE.W   JOY0DAT(a5),d0         ; Get current mouse counters
MOVE.B   d0,PrevH               ; Bottom 8 bits: horizontal counter
LSR.W    #8,d0                  ; Move the top 8 bits to the bottom
MOVE.B   d0,PrevV               ; Store the vertical counter value
```

The vertical blank interrupt will read the current counter values from JOY0DAT and use these with the previous values stored in PrevH and PrevV to calculate the relative change in both directions. These relative changes are used to update the X and Y scroll positions. These are stored in “ScrollX” and “ScrollY” respectively but inside the interrupt the registers D3 and D4 are used to store these two coordinate values.

```
MOVE.W   ScrollX(PC),d3         ; D3 - X Pos as calculated previously
MOVE.W   ScrollY(PC),d4         ; D4 - Y Pos as calculated previously
MOVE.W   $DFF000+JOY0DAT,d0     ; Current relative mouse position
MOVE.W   d0,d1                  ; Bottom byte: H, top byte: V
SUB.B    PrevH,d0               ; Subtract previous horizontal counter
EXT.W    d0                     ; Extend from byte to word size
ADD.W    d0,d3                  ; Calculate new X position
MOVE.B   d1,PrevH               ; Store counter value for next time
LSR.W    #8,d1                  ; Bottom byte: V
MOVE.B   PrevV,d0               ; Get the previous vertical counter
MOVE.B   d1,PrevV               ; Store the new counter for next time
SUB.W    d0,d1                  ; Calculate new Y position
EXT.W    d1                     ; Extend from byte to word size
ADD.W    d1,d4                  ; Calculate new Y position
```

From there on the interrupt routine will limit the values in D3 and D4 so that the bitplanes can’t be scrolled too far in either of the four directions. The X and Y coordinates then need to be converted into a delay value and a bitmap pointer position for each of the two bitplanes. This is done with the

If there is a CLI pointer in the process structure then the program was started from AmigaDOS and the Workbench related code must be skipped. The Workbench will send a message that needs to be stored so that it can be used to reply to the Workbench when the program is done.

```
            LEA.L    proc_MsgPort(a5),a0    ; A0 = Workbench MsgPort
            JSR      ExecWaitPort(a6)       ; Wait for Workbench message
            LEA.L    proc_MsgPort(a5),a0    ; A0 = Workbench MsgPort
            JSR      ExecGetMsg(a6)         ; Get Workbench message
            MOVE.L   d0,_S_WBMsg            ; Store message pointer
```

Replying to the message at the end of the program is done as shown below. If there is no message pointer then the program was likely started from AmigaDOS.

```
            TST.L    _S_WBMsg               ; Was there a message from Workbench?
            BEQ.B    .NoWBMsg               ; No. Nothing to do
            MOVE.L   _S_WBMsg(PC),a1        ; Pointer to Workbench message
            JSR      ExecReplyMsg(a6)       ; Reply message to Workbench
```

When the Workbench message has been received the OS can be stopped. This is done by calling the Exec library function Forbid(). This will forbid the task scheduler from swapping the current task for a different one.

```
            JSR      ExecForbid(a6)         ; Do not run other tasks
```

The only time when other tasks can still be swapped in is by changing the status of the current task to sleeping. This happens when calling one of the “Wait” functions of Exec, like the WaitPort() message used earlier. Please note that various dos.library functions will also put the current task to sleep, which could cause the OS to access the hardware unexpectedly.

The Forbid() state will automatically disappear when our process ends, there is no need to call Permit() at the end of the program.

In order to store the interrupt vectors used by the OS the routine first needs to find out where the vectors are located in memory. This location is stored in the processor’s VBR. Of the processors in the 68k family only the original 68000 does not have a VBR. Instructions that access the VBR are privileged and need to be called via the Supervisor() function in Exec as shown below:

```
            BTST.B   #0,exec_AttnFlags+1(a6); Check if > 68000 processor
            BEQ.B    .NoVBR                 ; On 68000 no VBR (always zero)
            LEA.L    _S_GetVBR(PC),a5       ; Function to call as supervisor
            JSR      ExecSupervisor(a6)     ; Call supervisor function in A5
            MOVE.L   d0,S_VBR               ; Store the returned VBR contents
```

The function that is called by Supervisor() copies the contents of the VBR into D0 and returns:

```
_S_GetVBR:  DC.L     $4E7A0801              ; MOVEC VBR,d0 (Place VBR contents in D0)
            RTE                             ; Return from supervisor mode
```

When the processor returns from the main program DMA and interrupts may have been enabled by the main program. Before restoring the original vector values it is important to ensure that no DMA or interrupts are left active.

```
LEA.L    $DFF000,a5             ; A5 = Chipset registers base address
MOVE.W   #$01FF,DMACON(a5)      ; Disable all DMA
MOVE.W   #$3FFF,INTENA(a5)      ; Disable all interrupts

MOVE.L   S_VBR(PC),a0           ; A0 = Pointer to vector base
MOVE.L   (a7)+,IRQ4(a0)         ; Restore IRQ4 vector
MOVE.L   (a7)+,IRQ3(a0)         ; Restore IRQ3 vector
MOVE.L   (a7)+,IRQ1(a0)         ; Restore IRQ1 vector
```

Pointers to the two coplist used by the OS are stored in graphics library and can be copied back into the two coplist pointer registers. It is important to make the Copper use the new value before restoring the DMACON, INTENA and ADKCON registers:

```
MOVE.L   S_GraBase(PC),a6       ; A6 = Graphics base
MOVE.L   gfx_copinit(a6),COP1LC(a5)      ; Restore coplist pointer 1
MOVE.L   gfx_LOFlist(a6),COP2LC(a5)      ; Restore coplist pointer 2
CLR.W    COPJMP1(a5)            ; Make Copper use restored pointer

MOVE.W   (a7)+,ADKCON(a5)       ; Restore audio, disk and UART
MOVE.W   (a7)+,INTENA(a5)       ; Restore original interrupts
MOVE.W   (a7)+,DMACON(a5)       ; Restore original DMA
```

The pointer of the original view is used to load the view again, which will restore the video signal to however it was set by the OS.

```
MOVE.L   (a7)+,a1               ; Get original view pointer
JSR      GfxLoadView(a6)        ; Restore the original view
JSR      GfxWaitTOF(a6)         ; Wait one screen refresh
JSR      GfxWaitTOF(a6)         ; Wait a 2nd (in case of interlace)
```

At this point all that is left is to reply to the Workbench message (if there was any) and to close graphics library, which was opened at the start. The last RTS will end the program, which will also end the current process. The end of the current process will allow other tasks to be scheduled in again. This allows the OS to continue from where it was interrupted.