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 15, typo: ECG/AGA should be ECS/AGA.

Page 19, typo: the phrase “new about this issue” should be “knew about this issue”.

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 98, Comments in Coplist example

The comments after the BPL2PTH and BPL2PTL refer to bitplane 1, which should be bitplane 2.

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 199, The TOD clocking

According to the datasheet the TOD is stopped when writing to TODMID or TODHI. However, Toni Willen (WinUAE developer) has found that only TODHI stops the clock, TODMID does not.

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.

For example, if the SET/CLR bit is bit 15 of the register and bits 3 and 2 are the bit of the register that need to be set then this is done by writing a word with bit 15, 3 and 2 set. If bits 2 and 3 of the register need to be cleared then this is done by writing a word with only bits 2 and 3 set.

Apart from the read-only and write-only registers there is a third category of registers, the strobe register. Accessing a strobe register will trigger something to happen. The actual value that was read from (or written to) the strobe register does not matter, only the action of accessing the register does.

Just like the databus of the MC68000, the registers of the chipset are all 16 bits wide. This is still true for the AGA chipset, even though that was only used with 680x0 processors that had a 32 bits wide databus. A register that is 16 bits wide is not wide enough to store a memory address. To store memory locations for things like audio data or sprite data two 16 bit registers are combined. The register ending in L will contain the lowest 16 bits of the address, where the rest of the bits will go to the register with the name ending in a H. The registers are always arranged in such a way that a 32 bits write to the "H" register will correctly fill the "H" register with the upper 16 bits and the "L" register with the lower 16 bits of the address. Please note that 8 bits writes to any of the registers (excluding BLTCON0L on ECS/AGA) is not supported.

Video modes

The OCS machines can display only one video mode; for example, systems sold in the USA only generate NTSC video and systems sold in Europe only generate PAL video. This changed with the introduction of ECS. Not only could European systems now produce an NTSC signal and US systems a PAL signal, the Amiga could now also support video modes like the 640x480 "productivity" mode, which is VGA compatible. To support this new functionality ECS introduced a number of registers that deal with the video timing. This functionality is used via the "Screen Mode" prefs program in conjunction with the monitor files present in the "Monitors" folder in "Devs:". This allows the user to select the resolutions and signals supported by the actual monitor used with their Amiga, which was important since it is possible for a CRT monitor to be damaged by trying to display an incompatible video signal.

The only two modes that can safely be assumed to work are PAL and NTSC, since these two are the only modes that OCS systems support. As a result games and demos that directly use the hardware will use either PAL or NTSC (or try to detect which one is appropriate for the system). Since this book is about directly using the hardware it will concentrate on PAL/NTSC and ignore the other modes.

Genlocks

It is possible to mix multiple analogue video signals, but for this to work each of the signals must be completely in sync with the other signals. being in sync in this case means that the horizontal blanks and vertical blanks all need to happen at the same time. For two independently clocked video sources to be completely in sync like that is extremely unlikely. Even when started at exactly the same time the signals will slowly start to deviate from one another due to small variances in the clock speed and within a very short space of time they no longer will be in sync with each other.

The designers of the Amiga knew about this issue and designed the video circuitry of the custom chips in such a way that an external video clock can be used instead of the Amiga's internally generated clock. This external clock then ensures that the video signal generated by the Amiga is fully in sync with another video signal, allowing both to be mixed.

The external video clock is provided by a device called a "genlock". This name is derived from the term "generator locking", which refers to the clock generators of the two video signals being locked together. The Amiga's capability of using an external video clock means that the device required to do the video mixing can be significantly less complex than it otherwise would have needed to be.

Genlocks on Amigas can be used for generating title sequences and sub titles, but also for more advanced techniques like replacing a colour in one video signal with the contents of the other video signal, a technique that is known as chroma-keying or green-screening.

Further discussion on the use of genlocks is out of the scope of this book.

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.

Vertical scrolling

This example shows how a playfield with bitplanes that are much taller than the display window can be scrolled vertically across the display window. In the case of this example the display window is 256 lines high and the bitplanes are 1024 lines high. The last 256 lines of the bitplanes are the same as the first 256 lines to allow for a simple continuous loop. Vertical scrolling is rather simple since it can be done by changing the bitplane pointer values. By increasing the pointer value by the amount of one line the bitplane will be scrolled up one line. Scrolling down can be achieved by decreasing the pointer values by one line.

To ensure smooth scrolling this example makes use of the vertical blank interrupt. This interrupt is generated during each vertical blank, which is ideal for making changes to elements on screen just before they become visible. And since the vertical blank interrupt is generated in sync with the video hardware it allows for the creation of smooth synchronised animation, like in this example where the scrolling happens at the speed of one line per vertical blank.

While the CPU is busy processing the vertical blank interrupt the Copper will be jumping back to the start of the coplist and start executing from there. This makes it very likely that the Copper will already have processed the little coplist by the time the processor gets to writing the newly calculated bitplane pointer values. The bitplane pointers must be valid when the display window starts, but do not need to be valid before that. The coplist is changed so that the first instruction makes the Copper wait for line $2A, which gives the CPU plenty of time to process the interrupt handler and write the new values to the coplist before the Copper gets to the instructions that write the bitplane pointers to the pointer registers:

```
Coplist:    DC.W    $2A07,$fffe           ; Wait for start of line $2A
CopBPL:     DC.W    BPL1PTH,0             ; High word APTR bitplane 1
            DC.W    BPL1PTL,0             ; Low word APTR bitplane 1
            DC.W    BPL2PTH,0             ; High word APTR bitplane 2
            DC.W    BPL2PTL,0             ; Low word APTR bitplane 2
            DC.W    $ffff,$fffe           ; Wait indefinitely
```

The initialisation of this example is the same as the one of the first example. The same size and position of display window is used, the same low resolution playfield with 4 colours. The bitplanes again have the same width as the display window and use a BPLxMOD value of zero.

The interrupt handler will first clear the vertical blank flag in the INTREQ register and then store some processor registers on the stack so that these can be used within the handler:

```
IRQHandler: MOVE.W   #$0020,$DFF000+INTREQ  ; Clear the interrupt flag
            MOVEM.L  d0-d2/a0-a1,-(a7)      ; Store d0,d1,d2, a0 and a1 on the stack
```

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

Timer A can be configured to use one of two sources as the input for its counter. This is done with the INMODE bit of the CRA register. When the INMODE bit is set timer A will use a positive edge on the CIA's CNT pin as its source. When the INMODE bit is cleared it will use the pulses on the O2 clock, which is the Amiga's system clock divided by 10.

Timer B can be configured to use one of the same two sources as timer A, but is also capable of using the output of timer A as a source. Using the output of timer A as a source to timer B allows timer A to be used as a pre-scaler for the input of timer B. Input selection for timer B is done via the CRB register, which provides two INMODE bits for timer B.

The TOD clock

The 8520 CIA also contains a 24 bits time-of-day clock with alarm function. The 24 bits are divided into three 8 bit registers named TODLO, TODMID and TODHI. As the names imply, the TODLO register contains the least significant byte and the TODHI contains the most significant byte, with the middle byte going to the TODMID register. Since writing a new value to the clock happens in 3 separate writes it is possible for the clock to change while writing to it. For this reason the CIA will pause the clock when the any of the TOD registers is written to and start the clock again when the TODLO register is written. Writes to the TOD registers therefore always must end with a write to TODLO, which will directly start the clock with the newly written value. Please note that according to the 8520 datasheet writing to either TODMID or TODHI will stop the clock. However, Toni Willen (WinUAE developer) has noted that actually only a write to TODHI will stop the clock, while writing to TODMID does not.

The same problem can cause misreads when reading the clock value while the clock is running. Instead of pausing the clock with each read the CIA will copy the values of all three registers into latches when TODHI is read. This keeps the value from changing while the three registers are read. Only after the TODLO register has been read will the CIA clear the latches so that a subsequent read will not read the same value again. For this reason the TOD registers must be read in the order of TODHI, TODMID and finally TODLO.

The alarm function of the TOD clock can be accessed by setting the ALARM bit of register CRB. When the ALARM bit is set writing to TODLO, TODMID and TODHI will set the alarm event value. When the TOD clock reaches this alarm event value the CIA will generate an ALRM interrupt. It is not possible to read the current alarm event value from the CIA; reading from TODLO, TODMID and TODHI when ALARM is set in CRB will still yield the current TOD clock value and not the alarm event value.

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.