Programming in machine code on the Amiga

A. Forness & N. A. Holten

ISSUE 2

Content

DATA SCHOOL
Postbox 62
North Engen 18
2980 Kokkedal

Phone 49 18 00 77
Postgiro 7 24 23 44

DMA CHANNELS

The Amiga has a number of DMA channels. "DMA" stands for "Direct Memory Access." These channels are used by the three COPROCESSORS: Agnus, Paula and Denise. A coprocessor writes to (and reads from) memory without the help of CPU (MC 68000). This will only be used to configure and start DMA. Let's have a look at an example.

This example is a simple and imperfect illustration of how DMA works. The full explanation, you get in a subsequent section.

We want to read from a disk. The data from the disk is loaded into a certain area of chip-ram . The CPU (68000) defines the address of this area which called the disk buffer. We must define with how many bytes we want to read, start the disk motor and finally start the disk-DMA (ie put the co-processors start disk operation).

disk-DMA will read from disk and store the data itself in our disk buffer. The main processor (MC-68000) meanwhile is free to continue with something else.

ATTENTION! The data that have been read from disk are stored in chip-ram. Do you remember why? Because it is Agnus, Paula and Denis, performing DMA-operations, and they can only use the memory, which ranges from $000000 to $07FFFF (the so-called chip-ram). See issue I.

When disk-DMA is finished reading, it will send an interrupt signal to the MC-68000, which e.g., enables the CPU to react immediately on the finished disk-DMA and continue its interrupted task after the handling of the disk data was done.

Amiga DMA channels:

TYPE	Number CHANNELS
disc	1
audio	4 (four audio)
sprite	8 (eight Sprite)
bitplane	6
copper	1
blitter	4

In total there are 24 DMA channels on the Amiga. All DMA channels can be started and operated simultaneously, if it is necessary.

You've probably seen your Workbench screen many times without having thought about what actually happens when it is there. It's more than you think! We just give an overview now, but later we will go in more detail. So don't worry if you don't understand everything at the beginning.

The Workbench-screen uses the copper-DMA channel and 2 bitplane DMA channels. With 2 bitplanes we have a maximum of 4 colors on the Workbench screen. (the concept of bitplanes will be explained in detail later). The cursor - the red arrow you can move around on the Workbench as with your mouse, uses a sprite-DMA channel. When you open or move a window (or icon) up to 4 blitter-DMA channels are used. Menus in the window also use blitter-DMA. When reading from or writing to the disk, the disk-DMA channel is used. The CPU (68000 processor) is probably the one that has at least to do thanks to all DMA channels. Consider what a sad machine the Amiga would be if it did not have the 3 custom chips: Agnus, Paula and Denise.

DMA TIME SLOT ALLOCATION PER HORIZONTAL LINE

This headline is to get hiccups. Finding a good direct translation is difficult. It should be translated as "DMA time per horizontal line" or a little more freely translated: "How much time it takes to draw a horizontal line, divided into smaller time periods and how time is used by the Amiga's different (co)processors". Once you have read this chapter you will understand better.

But first let us take a digression from the Amiga and explain how a or a television screen – works in principle.

Inside the monitor, there is an electron-cannon which emits a video beam – a beam of electrons. In a color monitor or color television there are up to three of them - one for each of the basic colors RED, GREEN and BLUE. This is the electron beam, which represent what you see on the screen. The beam runs across the phosphor coated inside of the monitor or the television. When this layer with phosphor is hit by electrons it is lighted up for a short time on the little spot where the electron hit the screen.

With the three basic colors it can show all the amazing colors you see in a natural program on TV. The picture is not a fixed image that stays all the time. It will be drawn up 50 times per second - but for your eyes it is almost impossible to see it flicker. You can, however, may see the flicker easier with certain color combinations.

The European Amiga can display a screen with 313 total lines. The electron cannon starts with drawing line number 1 from left to right. Then it jumps back and draws line number 2. Then it continues with line 3, 4 and 5 etc. until it filled the entire screen. Then it jumps up from lower right corner to upper left corner and begins to draw the screen again. And as I said, this is done very fast - 50 times per second!

As a curiosity we mention that each line is 0.000064 seconds - or 64 micro-seconds (millionths seconds). During the time it takes to complete each line on screen - the 64 micro seconds - a lot things happen in the Amiga. It is just not so easy to understand but read it a couple of times, so you thoroughly understand it.

At the end of the chapter there is a sheet with numbers (FIGURE 1) which illustrates how a line on the screen appears and what is happening - and can happen – during the line is drawn. Or with other words: what the different DMA channels can do in the time it takes to draw a line on the screen. This is what the title of this section describes.

What you now have to learn is not that easy to understand. Some of it you will most likely not completely understand before you came further in the course. However, it is very important information and you should put some effort in understanding the topic as much as possible. Figure 1 is very useful when you program yourself, so even if you don’t understand everything now you will have many opportunities to clear it up later.

You only need to understand the concept in principle. We will gradually return to full details of sprites, bitplanes, sounds, etc. in the DISK OPERATIONS the section.

Look at Figure 1 and imagine that the line you see there, happening while the electron cannon draws a horizontal line. In other words: while the electron cannon draws a horizontal line, the Amiga performs what you see in Figure 1.

Disk DMA, sprite DMA and bitplane DMA always have a fixed "time slot" on the time line (or in this period) to carry out their tasks, while the blitter DMA, copper DMA and the CPU (MC-68000) have the "free slots" (the holes) which are left over. The white "columns" are the so called free clock pulses the blitter, copper and the CPU can also use. It should also be mentioned that if you do not use audio, these gray areas will be considered as white, and be available for the blitter, copper and the CPU instead.

MACHINE CODE-PROGRAMMING

Now we have a look at how machine code programming works. First, please note that from now on we will use the abbreviation MC for "machine code programming." Let us start by looking at two frequently used expressions in MC the WORD and LONGWORD. The former word “word” is a description for two bytes (16 bits). Therefore two Bytes bound together are called a word.

A longword is a composition of two words, representing 32 BIT (or 4 bytes if you like).

Let us take some examples:

Bytes:	$0A,	$5C,	$FF
Words:	$8020,	$FC9B,	$0001
LONG WORD:	$00DFF180,	$00BFE001,	$ 8102FA88

The CPU of the Amiga (MC68000) is a so-called a 16/32 bit processor. This means that the data bus is 16 bits wide. The English term BUS for data link can be explained as a bundle wires which the processors in the Amiga use to communicate with each other and with the memory. This enables the bus of the CPU to transfer 16 bits to and from memory (or other parts of the machine) at a time. The term 32 bit means that the processor can work internally with 32 bits simultaneously. The processor must be able to retrieve a longword (32 bit) and therefore must go out on the DATA bus twice (download two Words, 2 * 16 bit = 32 bit).

The processor has 16 registers. They are used to store instructions and data to be processed. Eight of these registers are called DATA REGISTERS (D0, D1, D2… D7). The eight others are named ADDRESS REGISTER (A0, A1, A2... A7). All registers except A7 (also called STACK-Pointer, SP); can be used freely by the user. We come back to the use of data register after the next section.

Let's look at some of the main machine code instructions:

MOVE, ADD, SUB, and LEA.

MOVE is used to move (copy) data from one location in memory to a otherwise. For example, from memory to a register, from a Register to memory or from one register to another range.

MOVE-instruction has many variations, including:

MOVE.B
MOVE.W
MOVE.L

The "B" after the MOVE means that we will have moved a number which has the size of a byte (8 bits). The "W" indicates numbers that have a word-size (16 bits). And you probably you've already guessed, that "L" means an instruction that affects a longword size (32 bits).

Also noticed that the move-instructions not really moves data to somewhere, the data will be copied to the new location. So: if one writes MOVE.L $05, D0 – then the longword which is found in the memory with address $000005 is copied into the data register D0. Then both the memory at address $000005 and the data register D0 contain the same longword.

The ADD instruction is used to add two numbers (add)
The SUB instruction is used to subtract a number from another (subtract).

Let us look at an example:

```
1       start:
2       move.l      #$50, DO
3       add.l       #$10, DO
4       move.l      #$100, Dl
5       move.l      #$5, D2
6       sub.l       D2, D1
7       RTS
```

Line 1: The word "start" is a LABEL. Without the colon ":" after the word the assembler tries to interpret the word as a machine code instruction. This is explained in more detail in a later section. At the moment it is sufficient to keep in mind that the assembler marks the address of the assembled code with the label in memory. Thereafter you can refer to the address in using the label instead of the address. In this example the label "start" is at the beginning of our program and therefore we can refer to the start address of our program using this label. So when you want to start your program from within K-Seka you just write: j start.

Line 2: Moves the constant number $50 into the data register D0.

Line 3: Adds the constant number $10 to the number in the data register D0.

Line 4: Moves the constant number $100 into the data register D1.

Line 5: Moves the constant number $5 into the data register D2.

Line 6: Subtracts the number which is in the data register D2 from the number contained in the data register D1 (Note: the result is in D1).

Line 7: This instruction means: RETURN FROM SUBROUTINE. This means that the program or the routine ends and returns to the calling instance – whichever instance it was (Workbench, CLI, or K-Seka).For instance when you start the program from within K-Seka with the "j start" command so the program will return to K-Seka after this RTS instruction was executed.

After the above program is complete, look at the registers' content:

D0 contains $60 ($50 + $10)
D1 contains $CA ($100 - $05)
D2 contains $05

Here we must discuss the MC68000 and some of its "addressing modes".

In one way or another we must of course have an opportunity to tell the CPU , how to treat the various numbers which have been assigned. This is done by writing data to be used in different ways.

Below you can see a few examples of this:

```
move.l    #10, D0
move.l    #$10, D0
move.l    $10, D0
```

The first instruction moves the decimal number 10 into register D0. This way of moving the numbers into a register is called "IMMIDIATE ADRESSING" and can be translated as "the immediate transfer of data."

The second instruction moves the hex-number $10 (hexadecimal) into Register D0 in the same way. The name of this data transfer is the same as in the example above.

The last instruction is moving a longword which starts at the memory address $000010 into D0. This is called "DIRECT ADRESSING".

You need not worry about what the different addressing modes are called. The important thing is that you will be able to see and understand the difference when you see the instructions. And believe us, you will be able to.

The character "#" indicates that a constant number is to be used. When the character "#" is missing in front of a figure, it means that the processor must retrieve a number from a memory address, and that number represents the address of the memory cell where the number is to be fetched from. This does MOVE.L $10, D0 as explained above: Go to memory-address $000010 and copies the longword to the register D0. This last instruction is called "LOAD EFFECTIVE ADDRESS. This is explained later".

There are several addressing modes (ADDRESSING MODES), but we will study them as we will need them later in this course.

Another interesting register of the CPU is the PROGRAM COUNTER (PC). This register contains the address of the next instruction/data that the CPU must fetch and perform/process. Each time it retrieves an instruction or data, the PC is increased, so it points to the download next location where to fetch instructions or data the next time. This register keeps track of the (address) location where the CPU reads and executes the instructions of the current program. You can use the program counter when you are programming, but we don't think to do it. There are other and better ways to use this register. More details on this later.

We take an example:

```
1       lea.l   copperlist, A1
2       RTS
3       copperlist:
4       dc.w    $2C01, $FFFE
5       dc.w    $00E0, $0000
6       dc.w    $00E2 …
```

Line 1: Load effective address (.1 = LONG WORD) of the copper list into the address register A1

Line 2: Return from subroutine. Go back to the place where the program was called.

Line 3: "copperlist" is not an instruction. This is just to indicate a point in the machine code program – we already explained it – it's a label. This label indicates the address of the first byte in copper list ($2C).

Line 4,5 and 6: Here begins the part of a larger program which should be performed by copper.

Programs that you write in K-Seka can be located anywhere in the memory where there is free space. The operating system in the Amiga provides this functionality automatically. We explain more on this later. Short the address that the label “copperlist” represents can be anywhere in the machine memory.

The instruction "LEA" in our example determines the address of the copper list by counting ahead of the instruction "lea" until the label "copper list. It figures out how many bytes it should move forward to find the label “copperlist”. The number is called an offset.

Imagine the program as a ruler and Instruction “lea” is 16 cm mark. Instead of saying that the copper list at 20 cm mark, we can say that it is 4 cm in front the position where the program is started just now.

This similarly works with "lea". The current position in the program can be found in the program counter - PC. What happens in reality is to take the address from the PC, adds the distance between the two (instructions and label), and then puts the result into A1.

Well: The distance between two instructions (or an instruction and a label) is called an offset. It is used to find the instruction address by adding to a different address. Do you understand? If not, read it again. It is easier than you think.

MOST SIGNIFICANT BIT & LEAST SIGNIFICANT BIT

The most significant bit or the MSB is the most important bit of a byte or data word. The least significant bit or LSB is the least important bit. For instance in the decimal number 5005 the number 5 to the left has the most value, while the number 5 on the right has smallest "value" - even though both number are the same. See also the section on the position of values in a subsection in "the hexadecimal number system". The following explains it further:

	MSB							LSB	
BIT NO.:	7	6	5	4	3	2	1	0	
BYTE:	1	0	1	0	1	0	1	0	(170 or $AA)
BIT VALUE:	128	64	32	16	8	4	2	1	

This shows a byte and its 8 bits. As you see, the bit no.7 values 128. The value 128 is the largest in the bit group and therefore we call bit no. 7 the MSB. Bit no. 0 has the value 1, and is the smallest in the bit group and therefore will be named the LSB. These definitions are usually used in tables or diagrams in order to facilitate the reference to the BIT, we are talking about. We will give you more examples of this in a later chapter.

STANDARDS OF MACHINE CODE OF K-Seka

Before we throw ourselves into the machine code programming, we have a look how the K-Seka is organized and how to write programs correctly. Below, we have a line with a typical machine code instruction. We will now show what the different parts of setup means.

```
FIELD 0     FIELD 1     FIELD 2     FIELD 3

23          move.l      $04, a6     ;EXEC address into a6
```

FIELD 0: In this box the assembler prints the line number. It makes it completely automatic, so you do not have think about it. Line numbers makes it easier for you to navigate through the source code.

FIELD 1: In this area the machine code instructions are written. Also labels are placed in this field.

FIELD 2: Here the data or orders are written for the instruction in field 1. A comma separates this field into two parts if necessary. The first part is called the operator and the second part is called operand. The left part can also be termed FROM-part and right part TO-part. In the example it becomes: Move the content FROM address $04 and put into address register A6.

FIELD 3: This field can filled with comments. For that the text you type is interpreted as a comment, you must start with a semicolon (;). If you do not, you get an error from the assembler when you try to assemble your program.

We usually use small letters and numbers when writing programs in K-Seka. One can very well use capitals (upper case) but we think it's confusing. You try of course and find out what suits you best.

In our examples, we will initially use both – the capitalization is used mark special things. Slowly but surely, we are moving to only use lowercase.

At the beginning of your programmer career path you should write many comments in your machine code programs. If you do not do it – it will be very time consuming later on to find out what your different routines are for.

After a while you'll probably (surely) have forgotten what all the different routines in your program do. A commentary-rich program is easy to read and understand. In addition, a second person can easily continue based on your ideas and develop your application further if he/she has the ability to easily sort out which parts of the program does what.

When we talk about routines, think about the following:

The program you write can be considered as a single large workshop, where each worker is specialized to perform a particular job. Let us for simplicity’s sake think a post office with such special-skilled workers. A person has task to open the post office every day. Another clear the mailboxes, a third sorts the letters, a fourth delivers it to the customers - and so on.

All who work at the post office have their own special tasks to perform. This is similar in your program. A program accepts the data you type on the screen. Another Part sorts them - some stores them - and so on – all according to what your program really needs to do.

These different parts of a program called the routine. The main program can jump to a routine that asks you (for example) to confirm that you want to erase anything on disk. When you verify that this is what you want - the program execution jumps back to the point where this subroutine, was called. When programming the MC68000 this is done using the RTS (return from sub-routine).

It is very important that this instruction is not omitted when you write your programs. And you must be careful that they are placed in the correct place, otherwise it breaks your program.

DO YOU HAVE EXTRA RAM IN YOUR AMIGA?

If you have additional memory in your Amiga, you must disconnect it to run the program examples in this course (except sample number 1 in the issue I). The reason is that if you write a program using the copper, the copper list is located together with the rest of the parts in FAST-RAM. As you probably remember the copper is part of Agnus, Paula and Denise, and they can only work in the chip ram. Of course there is another method but for simplicity's sake just remove the ram-expansion.

To disconnect the extra memory in addition, you can start up your Amiga with Workbench-disk. When the WB is loaded, open the System tray and start the program NOFASTMEM. Then go into the CLI. (K-Seka can only be started there).Then put the K-Seka-floppy in and type "df0: seka" (If you have an extra floppy station, you can write "DF1: seka").

COPPER

COPPER stands for COPROCESSOR (supportive processor).
The copper is a processor that has only 3 instructions (The MC68000 CPU has more than 100). The instructions for the COPPER are: WAIT (wait), SKIP (jump over) and MOVE (move data).

The copper can only be programmed to perform specific things. To write a program for the copper (called a "copper list"), we need to use the machine code instruction "DC.W" (English: define word, i.e. define two bytes). Let us explain using an example of a small assembler program:

The copper is mostly used with bitplanes (much more about it in issue III and IV), which are used to form the graphics screen.

```
1       move.w      #$01A0, $DFF096
2       lea.l       copper list, A1
3       move.l      A1, $DFF080
4
5       move.w      #$8080, $DFF096
6       wait:
7       btst        #6, $BFE001
8       BNE         wait
9
10      move.w      $0080, $DFF096
11      move.l      $04, A6
12      move.l      156 (A6), A1
13      move.l      38(A1), $DFF080
14      move.w      #$81A0, $DFF096
15      RTS
16
17      copper list:
18      dc.w    $9001, $FFFE
19      dc.w    $0180, $0F00
20      dc.w    $A001, $FFFE
21      dc.w    $0180, $0FFF
22      dc.w    $A401, $FFFE
23      dc.w    $0180, $000F
24      dc.w    $AA01, $FFFE
25      dc.w    $0180, $0FFF
26      dc.w    $AE01, $FFFE
27      dc.w    $0180, $0F00
28      dc.w    $BE01, $FFFE
29      dc.w    $0180, $0000
30      dc.w    $FFFF, $FFFE
```

Line numbers without text lines are only included for increasing the program's readability.

Line 1-4: Blank (cleans) the screen, stop the work bench, disable the sprite (red arrow) and start up our own copper list.

Line 6-8: This little routine is a loop (infinite circle) and waits for you to press the left mouse button when you want to quit the program.

Line 10 - 15: Retrieving the old copper settings back (as to reactivate the workbench) and start the copper.

Line 17 - 30: This is our own copper list, which forms a red, white, blue, white and red line.

There is a lot in this program, which needs a closer explanation. However, it is some knowledge we have not yet presented, so we choose to take this up again when we explains the bitplanes in issue III and IV.

There we explain this example in detail, also describing the instructions as RTS, and phenomena such as jumps elsewhere in the program (branching), bit-testing and the status-register.

As you may have the feeling we have only scraped the surface of all the opportunities the copper provides in programming context. Over the next two issues, as already mentioned, we explain bitplanes. We must also learn more details of the copper as well. It becomes easier for you to understand both bitplanes and the copper when we explain these two topics in combination.

SOLUTIONS FOR TASKS IN ISSUE I

Task 0101:

Decimal	Binary	Decimal	Binary
0	0000 0000	147	1001 0011
14	0000 1110	155	1001 1011
15	0000 1111	156	1001 1100
16	0001 0000	157	1001 1101
27	0001 1011	158	1001 1110
45	0010 1101	200	1100 1000
63	0011 1111	213	1101 0101
64	0100 0000	228	1110 0100
65	0100 0001	229	1110 0101
98	0110 0010	240	1111 0000
99	0110 0011	245	1111 0101
100	0110 0100	253	1111 1101
101	0110 0101	254	1111 1110
133	1000 0101	255	1111 1111

Task 0102:

Decimal	HEX	decimal	HEX
30	1E	255	FF
31	1F	256	100
32	20	257	101
33	21	2747	ABB
63	3F	2748	ABC
64	40	2749	ABD
65	41	4095	FFF
66	42	4096	1000
127	7F	4097	1001
128	80	4098	1002
129	81	4099	1003
130	82	5000	1388
170	AA	10000	2710
171	AB	20000	4E20
172	AC	43980	ABCC
254	FE	43981	ABCD

Task 0103:

Binary	HEX
0000001001101000	0268
1010101111001101	ABCD

HEX	Binary
ABCD	1010101111001101
FFFF	1111111111111111
D5C6	1101010111000110

TASKS TO CHAPTER II

Task 0201: Explain why we can state that the Amiga has 24 DMA-channels.

Task 0202: What is a subroutine in a program?

Task 0203: How many bytes are in a word, and how many bits?

Task 0204: How many bytes are in a longword and how many bits?

Task 0205: How many ADDRESS REGISTERS are there in a MC-68000, and what are their names?

Task 0206: How many data registers exist in MC-68000, and what are their names?

Task 0207: What does the following instruction do: MOVE.L $10, A3?

Task 0208: What is the MSB and LSB?

Task 0209: How many instructions has the copper and what are they?

Task 0210: What is the video beam and how is it generated?

SHORT DATA DICTIONARY

ADDRESS REGISTER	One of the 8 internal registers in the MC68000 - A0 to A7.
COPPER	Abbreviation for coprocessor. The copper has only three instructions. It is s often used in context with graphics.
DMA	"Direct Memory Access." A concept realized in the Amiga, to transfer data from memory without the help of other processors.
LONGWORD	A combination of 4 bytes (32 bit)
LSB	Least significant bit. Name of the rightmost bit of a bit-group that has the smallest position value.
MSB	Most significant bit. Name of the leftmost bit in a bit-group with the largest position value
PROGRAM COUNTER	Abbreviated PC. The register of the MC68000, which holds the address of the next instruction to be executed the next time.
SUB-PROGRAM	Subroutine in a larger program. Often a special way to perform a special task.
VIDEO BEAM	The beam of electrons produced by the monitor's electron gun which is concerned to draw an image on the screen, point by point.
WORD	A combination of two bytes (16 bits).

COMMENTS ON CHAPTER II

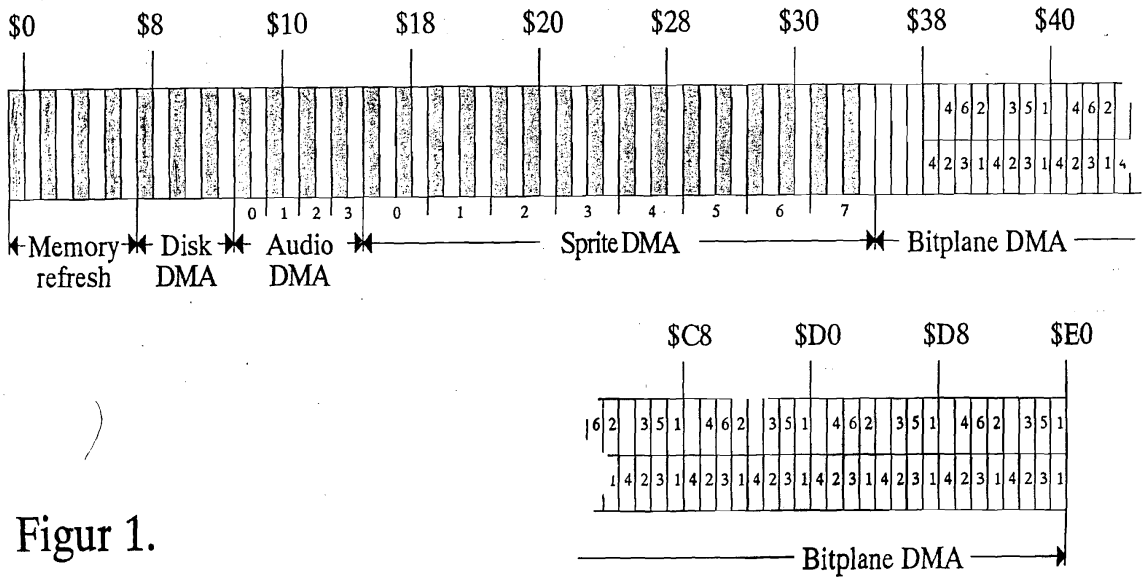
This was issue II. It was a little harder than the first issue, but it should not have been that complicated.

The tasks in issue II, as already mentioned the last time you may send to us and get a corrected version, but it is not necessary because the solutions to a chapter are always presented in the following issue. So in issue III you will find the solutions to the tasks in this chapter.

After the 20th of the month the next issue is "automatically" sent to you unless we do not receive your prepayment on attached giro. If you prepay you save on delivery fees and postage. In this way, the course is significantly cheaper for you.

Continue to enjoy!

DATASCHOOL
Carsten Nordenhof


$0
$8
$10
$18
$20
$28
$30
$38
$40
0 1 2 3 0 1 2 3 4 5 6 7
Memory refresh
Disk DMA
Audio DMA
Sprite DMA
Bitplane DMA

$C8
$D0
$D8
$E0
Bitplane DMA

Figur 1.