Programming in machine code on the Amiga

A. Forness & N. A. Holten

Issue 10

DATA SCHOOL
Postbox 62
Nordengen 18
2980 Kokkedal

Phone: 49 18 00 77
Postgiro: 7 24 23 44

LIBRARY AND SYSTEM FUNCTIONS

In this issue we deal with a part of the OPERATION SYSTEM in the Amiga, and we begin this chapter by explaining what a LIBRARY and a SYSTEM FUNCTION is.
The word LIBRARY means a collection ob books – in this case collection of useful sub-routines or function. The AMIGA makes use of many different libraries. Each of them contains a series of functions called SYSTEM FUNCTIONS. For instance the GRAPHIC library contains all the functions that have to do with graphics, while the DOS library contains all the functions that have to with disk management and Amiga DOS.

We explain now how to make use of libraries in your programs.

Have a first look to FIGURE 1 at the end of this chapter. This figure shows a draft of how a library might look like. At address $11000 is the BASE of the library. From address $11000 and above are the functions (or subroutines / programs), while "below" address $11000 is a series of pointers that point to the various functions above the library base.

So: When you need to jump to a function (routine), you will find its address by following one of the pointers - you do not directly jump to the routine you want to use.

Now you're probably ask yourself- “BUT WHY NOT?”. Well, when there is a new version of the operating system (Kickstart), program routines in the libraries are subject to change (and thereby hopefully improve). Routines are likely to change in size this leads to the fact that routines (functions) are located at other addresses than they did before.

So if you have gone directly to the routines in your programs – you have a potential problem. They might not work under the new operating system versions. So when the COMMODORE programmers are changing a library they also changing these pointers, while the list of pointers is always on the same place (relative to the library database).

We go a step further and show how to find the library base (address) of the so-called EXEC LIBRARY and it is simple as follows:

```
MOVE.L $4, A6
```

We've now got the address of the EXEC-BASE in the address register A6. The AMIGA always stores the address of the EXEC-BASE at this address - $04, when the machine is switched on. This is the only guaranteed absolute address in the Amiga. We now know that at the addresses below the address is located in the A6, are the points. We need not to deal with more than one function of this library right now. This feature is called "Open Library", and the pointer is in OFFSET 408 under / below base. A jump to this function will therefore look like this:

```
JSR -408 (A6)
```

This function opens a library. We did not yet talk about which library is to be opened. Here is a program that opens the DOS library:

```
        MOVE.L      $4, A6
        LEA.L       dosname, A1
        JSR         -408 (A6)
        RTS
dosname:
        DC.B "dos.library", 0
```

The first line moves the EXEC-BASE into A6.

The second line loads the effective address of "dosname" into A1. This register now points to a text which we have declared in the bottom of the program. Notice that we put a zero in the last letter of the name. This must be done that so the function (Open Library) must be able to know where the end of the defined text is.

The third program line jumps to the "Open Library" sub routine. When routine ("Open Library" function) is executed, the BASE address of the dos library can be found in D0.

So: "Open Library" function will have a pointer to the library-name in A1, and return the BASE address of this library to D0. If the "Open Library" function ins not able to open the desired library, a 0 value is returned in D0. In this way you can compare D0 with 0 to check whether the opening was successful or whether there was an error.

Notice that all system functions which return a value do this in register DO. Also note that the BASE address to the library you need to work with, always should be in the A6. To clarify it use the A6 to work with the DOS library. You can put this instruction after the JSR instruction in the program above:

```
MOVE.L      D0, A6
```

The next "function-call" you do after this instruction will now be in the DOS library.

We have now given you a simple introduction to how libraries (Library - plural: LIBRARIES) built and how they are used with the AMIGA. In the next chapter we will continue with two other features from EXEC-Library.

MEMORY ALLOCATION

In this chapter we learn about memory-allocation (how you reserve memory for special purposes) on the Amiga.

Imagine that you have written a program which sets up a blank screen with a size of 320 * 256 pixels in an BITPLANE. To display this screen, you need 10,240 BYTES memory (320 * 256 = 81920 PIXELS, 81920 / 8 = 10240 bytes, 10240/1024 = l0kB). You must therefore set up a buffer in the program with this size. The downside of putting this buffer (block) in the program is that the executable file (Object File) will contain 10 KB with only zeros. This makes the file unnecessarily large.

Another method of doing this is to allocate memory for the screen. Allocate actually and means "reserve". The AMIGA operating system can handle multiple programs at a time – which is called multi tasking - you can not use any absolute address for your display memory since we never know whether it’s free or used by another program. Therefore we must use a system function to find out where a block of free space in memory can be found and allocated. So you declare how many bytes you want and what type of memory it must be (CHIP or FAST). Then you'll jump to the system function which returns the starting address of memory block (the place in memory where of the byte of your reserved screen memory).

If there is not enough free memory in the return value in d0 is 0.

Once you have allocated a memory block, this memory area is protected against another allocation by a different program. We must therefore release the memory block before we exit the program. If we don’t release the memory less and less memory will be available if we run the program several times. You may eventually be forced to perform a RESET (Boot up) to get the machine's memory freed again.

Let us look at a program example MC1001 (which is on course disk No. 1 - AMIGA MC DISC 1), where we allocate a memory block of 100000 bytes.

Line 1: Adds the value 100000 into D0.

Line 2: Jumps to the sub-routine "allocchip".

Line 4: Compares the value in D0 with value 0

Line 5: If D0 is equal 0, jumping to "nomem". This means that there are NO 100000 BYTES free of CHIP memory ➔ the program will jump to "nomem".

Line 7: Loading the effective address of the "buffer" into the register A0.

Line 8: Move the value, which is in D0 to the address A0 points to. In the program line 22, we have allocated a long word to put the starting address of the just reserved memory block into.

Line 12: Moves the constant value 100000 into D0.

Line 13: Loads the effective address of "buffer" into A0.

Line 14: move the value A0 (i.e. "buffer") points to to A0 again. The A0 register will now contain the start address of the memory block.

Line 15: Branches to the sub-routine with the label "freemem".

Line 16: Ends the program.

Line 18-19: This label is defined and jumped to if there is not enough memory available.

Line 21-22: Here is the long word defined to store the starting address of the memory block.

Line 25-32,
34-40,
42-48:

These routines are quite similar, except that they allocate different types of memory. The first routine (allocdef) allocates FAST memory if it exists, if not it allocates CHIP memory.

The second routine (allocchip) will always try to allocate CHIP memory.

The third tries to allocate only FAST memory.

We explain only the first of these three routines, as they are almost identical.

Line 26: Saves all the registers except D0 to STACK. D0 not stored, because it is used to return a value.

Line 27-28: Moves quickly the constant value 1 into D1 and swap the high- and low-word in D1. Register D1 will now contain $00010000 – which indicates the kind of memory which is to be reserved.

Line 29: Moves the value (exec base) located at address $000004 into A6. A6 will now contain the base address of exec library.

Line 30: Jumps to the system function "AllocMem". The "AllocMem" uses the content of D0 as the size of the memory and the content in D1 as the type of memory to allocate.

- The value of $10000 in Dl indicates that FAST memory (if it exists) is allocated. If not FAST memory is available then CHIP memory is allocated.
- The value $10002 will lead to an allocation of CHIP memory.
- The value $10004 will allocate FAST memory. As mentioned earlier that function returns the memory block starting address in the D0 register. If there is not enough memory, it returns 0 in the D0 register.

Line 31: Restores the saved register values from the stack.

Line 32: Ends the routine, return to the calling instance.

Line 50: Here begins the routine that frees a memory a block after it has been allocated.

Line 51: Saves all the registers to the stack. Notice that this routine does not return any value.

Line 52: Copies A0 into A1.

Line 53: Moves the EXEC-BASE into A6.

Line 54: Jumps to the system function "FreeMem". This function will free the memory, which has been allocated before, using the start of the memory block in register A0, as well as the size in register D0.

Line 55: The registers are restored from the stack.

Line 56: End of the routine, return to the calling instance.

We have now looked at how to allocate memory on the Amiga. This program (MC1001) does not use the memory which was allocated. It frees it immediately again after allocation. It is intended that you should use these routines in your own programs.

READING AND WRITING FILES

In this chapter we will look at how to read and write files on a disk.

It is very useful to be able to handle files when you are programming. System functions for this are located in the DOS library and have names like: OPEN, CLOSE, READ and WRITE. Before we can come to either read a file or write to it, the file has to be opened. This is done with the OPEN function. Then you can use the READ and WRITE function to read or write data from and to the file. When you are finished processing the file, it must be closed. For this the CLOSE function is used.

Let us begin with the program example MC1002. It reads a file - Called "testfile" - into "buffer".

Line 1: Move the constant value of 24 into the register D0. This value specifies the maximum length (number of bytes) to read. In this case, we know that the file has 24 bytes long, but you can set this value higher if you want (remember to expand the size of the buffer).

So: This value is set to the maximum number of bytes o load. If the file is shorter than the specified value will the entire file will be loaded. If the file is longer than the specified value, only the specified number of bytes is loaded. So beware that you get the entire file - if that is what you want.

Line 2: Loads the effective address of "filename" into A0.

Line 3: Loads the effective address of "buffer" into the A1.

Line 5: Branches to the sub-routine "read file".

Line 7: Compares the value in D0 with value 0.

Line 8: If D0 is equal 0, branch to the label "error".

Line 10: Ends the program.

Line 12-13: If there is an error after loading the file some “error handling” can be placed here before exiting the program.

Line 15-16: Here the name of the file to be loaded is declared.

Notice that the name must end with a trailing 0 (zero).

Line 18-19: Here are block of 50 bytes is reserved for storing the data from the file.

Line 22: Here begins the routine that loads the file.

Line 23: Saves all registers - except D0 – to the stack. D0 is not stored, because the register returns a value.

Line 24: Copies the value in A0 to A4.

Line 25: Copies the value in A1 to A5.

Line 26: Copies the value in D0 into D5.

Line 27: Moves the exec-base into A6.

Line 28: Loads the effective address of the DOS library's name "r_dosname" to A1. A1 now points to the text "dos.library".

Line 29: Jumps to the "Open Library" function.

Line 30: Moves the content from D0 into A6. This register now contains address of DOS library (DOS-BASE).

Line 31: Moves the value 1005 into D2. This value means to open an existing file (MODE_OLDFILE).

Line 32: Copies the value in A4 to D1. The register now contains the pointer to the file name.

Line 33: Jumps to the "Open" function.

Line 34: Compares the value in D0 with 0.

Line 35: If D0 is 0, there was an error opening the gile therefore branch to the label "r_error". So if the file was not found it will branch to "r_error".

Line 36: Copies the value in D0 into D1.

Line 37: Copies the value in D0 into D7.

Line 38: Copies the value in A5 into D2.

Line 39: Copies the value D5 into D3.

Line 40: Jumps to the DOS library function "Read".

Line 41: Copies the value in D7 into D1.

Line 42: Copies the value in D0 into D7.

Line 43: Jumps to the DOS library function "Close".

Line 44: Copies the value of D7 into D0.

Line 45: Restoring the register values from the STACK.

Line 46: Ends the routine, return to the calling instance.

Line 47-50: “Error-handling” if the file does not exist.

Line 52-53: Here is the name for the DOS library declared "dos.library".

Calling the routine "readfile" the following register are expected to have follwing values:

D0 = Maximum bytes to read (maximum length).

A0 = Pointer to filename.

A1 = Pointer to buffer where to file data is to be stored.

When the routine "read file" is executed, it returns the length of the byte which were read in the register D0.

To run this program you must be in the directory "BREV10" on the course-disk. It can be done from K-Seka with the following command:

We assume that you have placed the course-disk in drive "dfl:".

Seka> vdf1: brev10

You will now get a list of all files located in the directory "BREV10". Among all files there must be a file called "testfile". Assemble the program and start it with the command "j".

To see if the file is really loaded, you can do the following:

SEKA>qbuffer

You will now see the memory where the "buffer" is located and the data of the file was read to. Was it okay? We hope so, and will start the explanation for example MC1003. This program is almost identical to the MC1002, so we mention only those things that are different.

The meaning of this program example is to create a file that is called "testfile" and then write the text "Hello, this is a test!" into it.

The definition of the routine "writefile" looks like this:

IN Parameter:
D0 = length of to be written.
A0 = Pointer to filename
A1 = Pointer to data to be written.

OUT Parameter:
D0 = length which was written.

The routine returns the length which was written to the file in D0. You can compare D0 with 0 to find out whether there were errors in writing. If D0 is equal to the value D0 had before calling "writefile" the routine was successful.

We think there is no need for further explanation of this program example. The important thing is to not to fully understand how the routines (allocchip, writefile, etc.) are programmed but how you use them!

ARGUMENT-LOADING

In this section we look at ARGUMENTS.

Let us first take an example. You've probably used the command "list" sometimes. This command is available on all-WORKBENCH disks in DIRECTORY "c". This command is really a fairly common program. When you type "list df0:", the "list" program will be performed while "df0" is a supplementary information, which the program needs to perform its job - and this is called an argument.

If we created a program that loads a file from disk and then write its contents to the screen, it would be nice if we could give the file name directly in this way in CLI:

1> myprog testfile

This method we can use in our programs. Behind the scenes it is really quite simple. When you start your program (OBJECT-file) from the CLI, the register A0 contains the address of the location of the Arguments in memory, while the register D0 will contain how many characters it consists of (length of the ARGUMENTS).

Let's take a look at - and explain - program example MC1004.

Line 1: D0 compares with values 1 Notice that this is the length of the arguments +1 (plus one), as presented in register D0. So: If the argument is 4 characters long, D0 will contain 5.

Line 2: If D0 is equal to or less than one, jump to "noarg" this means if no arguments are passed jump to the label "noarg".

Line 4: Loads the effective address of "argbuffer" to A1.

Line 5: Copies the value in D0 to D7.

Line 8: Copies the values A0 points to, to the address A1 points to. After that both addresses in the registers (A0 and A1) are increased by one to point to the next byte.

Line 9: Decreases register D0 by 1

Line 10: Compares the value in D0 with 0

Line 11: If D0 is not equal 0, jumping to the label "copyarg".

So th program lines 8 to 11 copies the ARGUMENT text into the "argbuffer".

Line 13-16: These program lines write the ARGUMENT text to the screen (in the CLI window). The explanation how to write text to a CLI window can be found later in this issue.

Line 18: Ends the program, return to calling instance.

Line 20-21: If there no argument, jumping to this label – some message for the proper usage of the program or error handling can be placed here.

Line 23-24: Here the ARGUMENT buffer declared.

Line 26-54: This program lines write characters to the CLI window and will be explained later in this issue.

This program must be run outside of K-Seka otherwise it has no sense (you do not see much of CLI display in the K-Seka).

The example program also exists as an executable object-file on the course-disk and can be started as following:

```
1> MC1004 Hooray!
Hooray!
1>
```

You have now learned how to read in an argument which can be passed via the CLI after the program name. In the next chapter, we combine this function of argument passing with loading a text file and our text-scroll program.

SCROLL PROGRAM WITH TEXT FILE

In this chapter we look at a program MC1005 using some of the system functions we already have explained now.

The program example MC1005 is actually an expanded version of the program example from issue 7 MC0701 (SCROLL-text). The extension is that it loads the scroll-text from a file. It gets the file-name from the CLI as an argument from the starting command. The program provides an allocated buffer where the file is read into.

The code was already explained. Here only the program lines are explained that are new compared to the example MC0701.

Line 1: Compares the value in D0 with values 1.

Line 2: If D0 is greater than 1, the program jumps to the lable "argok". So if there is an argument that jump "argok".

Line 4: If there is no argument end the program.

Line 7: Load the effective address of "filename" into A1.

Line 9: Here begins the loop for copying the argument into the "filename buffer.

Line 10: Copy the byte A0 points to the address A1 points to and increase both addresses in the registers by 1.

Line 11: Subtracts 1 from the value contained in D0.

Line 12: Compares the value in D0 with the value 1. D0 is compared to 1 because at the end of the argument is an ASCII code that indicates the "line break" or "return". This character must not be copied into the "filename" buffer and therefore, we compare with 1 to skip over the last character.

Line 13: If D1 is not 1, the program jumps back to the label "copyargloop".

Line 15: Moves the value 50000 to D0.

Line 16: Branching to sub-routine "allocdef" who will try to Allocate 50000 bytes.

Line 18: Compares D0 with value 0.

Line 19: If D0 is different from 0 allocation was successful and jumps to "memok".

Line 21: If D0 was 0, memory allocation was not successful (e.g., there was not enough free memory) the program is ended and returns to the calling instance.

Line 24: Loads the effective address of "buffer" into the A1.

Line 25: Copies the value in D0 to the address where A1 points to (into "buffer"). It at the label "buffer" a long word was reserved and will now contain the starting address of the memory block that was allocated.

Line 27: Loads the effective address of "filename" into A0.

Line 28: Copies the value in D0 to A1.

Line 29: Moves the value 50000 into D0.

Line 31: Branch to the sub-routine "read file". This routine will now load the file "filename" into the allocated buffer with a maximum length of 50000.

Line 33: Compares the value in D0 with 0.

Line 34: If D0 is equal 0 (reading went wrong) it jumps to "freeup". If there was an error loading the file at the label "freeup" the allocated memory has to be freed before the program ends.

Line 36-77: These lines are identical with those in program example MC0701, and therefore are not explained here.

Line 79: Here the allocated memory is freed.
Line 80: Moves the value 50000 into D0 (size of the memory block to be freed).
Line 81: Loads the effective address of the "buffer" into A0.
Line 82: Moves the address (stored in the buffer) A0 points to A0 – so the the address of the allocated memory-block is in register A0.

Linie83: Branch to sub-routine "freemem".

Line 85-162: These program lines are already explained in program example MC0701.

Line 164-195: This routine is already explained in an earlier chapter of this issue. It is the routine that loads a file.

Line 197-212: These two routines allocated and free the memory and also have been explained in an earlier chapter in this issue.

Line 214-330: These lines are the same in MC0701 and need no further explanation here (at least we hope ...).

Line 332-333: Here is a long word is defined to store the starting address of the allocated memory block.

Line 335-336: Here we have reserved 50 bytes, where the filename is stored from the passed arguments at the program start.

To run this program you can use the already assembled object file located on the course-disk. Boot the course disk and type the following:

```
1> Brev10/MC1005 Brev10/Text
```

As you have notices in the directory of this chapter "BREV10" there is a small predefined text file. If you want to make your own text, so you can use a simple text editor such as Notepad or "ED". Remember to put a "*" character (an asterisk) at the end of the text, the SCROLL program can find out a, where the text ends.

Good luck!
In the next chapter, we learn something about directly reading from and writing to a floppy disk.

READING AND WRITING DIRECT TO DISK

In this chapter we review reading from and writing to disk directly at sector-level. We start by explaining how data is stored on a floppy disk. Have a look at FIGURE 2 at the end of the isue. An Amiga floppy disk is divided into 80 tracks - like ripples in water. Both sides of the disk are used to store data. Therefore we can say that a floppy disk has 160 tracks. These tracks are divided into smaller parts -sectors. Each track has 11 sectors. This is therefore 11 * 160 = 1760 sectors in total on an Amiga floppy disk. Each sector contains 512 bytes. The total storage capacity is therefore is 11 * 160 * 512 = 901,120 bytes (i.e. 880KB).

To find the location on the disk, where we have to read/write we need the track, disk-side and the sector. To make it simpler the Amiga uses the term BL0CK (also called data block - not to confuse this with a memory block) to specify a location on disk.

Let us give some examples:

TRACK	SIDE	SECTOR		BLOCK
0	0	0	=	0
0	0	1	=	1
0	0	2	=	2
0	0	10	=	10
0	1	0	=	11
0	1	10	=	21
1	0	0	=	22
1	0	1	=	23
10	0	0	=	220
40	0	0	=	880
40	1	0	=	891
70	0	0	=	1540
79	0	0	=	1738
79	1	0	=	1749
79	1	10	=	1759

As you see an Amiga floppy has in total 1760 BLOCK (numbered 0 to 1759). The block 0 and 1 are called BOOT-BLOCK. When you turn on the machine and put a disk in these two blocks are automatically loaded into memory (more on this in issue XII). At block 880 (track 40) and a little beyond (depends on how many files as possible on floppy disk) is the Directory. The word DIRECTORY can be translated with "book" or "contents". It is here so the AMIGA can find out what files exist and where the data in a file is located on disk.

Let us now look at a program example. MC1006: We will not rack our brains to explain the routines that read/write to the floppy. It is more important to learn how to use these routines.

Here come the registers which are used for passing parameters to the routine:

IN: A0 = Pointer to read or write buffer.
D0 = Disk Station.
D1 = Start-BLOCK.
D2 = Length (number of BLOCKS).
D3 = Mode.
1 = READ
2 = WRITE (buffer to disk)
3 = UPDATE

As you see the routine returns nothing. A0 must therefore contain the address of the buffer, which must be read from or written to.

The value in D0 determines what floppy station is to be used. You specify a number between 0 to 3 which indicated "DFO", "DF1", "DF2" or "DF3".

The value in Dl determines the starting block, and the value in D2 determines the length (in blocks), which is be read or written.

The value in register D3 determines whether you need to read or write. The value 1 means loading, value 2 means writing. The value 3 is rather special. Let us first tell you that the AMIGA is using a so-called disk buffer. Imagine that you entered block 0 and then have changed a little on that data. Then write it back to disk (using mode 2). You will then discover that the disk station does not respond. This happens because BLOCK 0 lies in the disk buffer (in memory). When you write the BLOCK the AMIGA puts the data in disk cache, and not directly on the floppy. Therefore you first use mode 2 for new data in disk buffer. Then you use mode 3 to update the disk with new data which is now in disk buffer.

We continue with a test of this program example. The first you have to do is to format a new disk. If you use a diskette with files, these files could be lost. Therefore, use a new and empty disk.

When you are finished formatting the disk you want to use, start K-Seka and load the example application into the editor. Then alter your program as follows:

```
lea.l       buffer a0
move.l      #0, d0
move.l      #100, d1
move.l      #1, d2 (continued on next page)
```

```
        move.l      #2, d3
        bsr         sector
        move.l      #3, d3
        bsr         sector
        rts
        .....
buffer:
        dc.b        "This is a test"
        blk.b       512,0
```

Once you have changed this program, you can assemble it. Insert a formatted floppy disk into "DF0" and run the program. If everything went well the drive's led lights up for about 1 second.

So we have written the data out to disk. To load the block again changing your program as follows:

```
        lea.l       buffer, a0
        move.l      #0, d0
        move.l      #100, d1
        move.l      #1, d2
        move.l      #1, d3
        bsr         sector
        rts
        .....
buffer:
        blk.b       512,0
```

Assemble the program and run it. To check the data read from disk type the following:

```
        Seka> qbuffer
```

You certainly did notice that the drive's led was not lightning up. It is because the block we just wrote was still in the disk buffer. Why doesn't the Amiga obtain the data from the drive?

Try to take the disk out and put it in again. Then run the program again (assemble it first). Now you will see that the disk drive starts up. We have now looked at how you can read and write directly to a disk-block (sector).

In the next chapter we see go into details on how to write to and read from the CLI window.

READ FROM AND WRITE TO CLI

In this chapter we review the input and output to the CLI window. Reading and writing (INPUT and OUTPUT) to the CLI window is done almost the same way as reading from and writing to a files. But Rather than opening a file, we just use two other functions to input and output. Open is not really needed here since the CLI window is already opened. If we used the OPEN function the AMIGA would open a new window on the screen. This is not what we intend when writing to the CLI window.

We have made complete routines for both reading and writing to the CLI window. Here's the setup for these two routines.

readchar:
IN: A0 = Pointer to a buffer.
D0 = Maximum read length.

OUT: D0 = current length.

writechar:
IN: A0 = Pointer to a buffer.
D0 = Length.
OUT: D0 = current length.

For routine "readchar" (read characters) A0 must contain the address of the buffer, which the character data (keyboard data) is to be read in. Register DO represents the maximum amount of data you want to read in. D0 as well returns the actual length, which was read. An input is terminated by pressing the "RETURN" or by reaching the maximum number of entered characters. Notice that the characters which are read in are also shown on the screen (CLI window).

In the second routine called "writechar" (write character) A0 contains the address of the text to be printed out.

The register D0 must contain the length of the text, and returns the length (the number of characters) which was written.

In the program we have a routine which we have not mentioned yet. It's called "opendos", but the only thing this routine does is to open the "dos.library" and then store the dos-base in the longword which is reserved at the end of the program (at the label "txt_dosbase"). This routine must be called only once at the start of the progeam. It must be called before you can use the sub-routines "readchar" and "writechar".

Now let’s have a closer look at a program example MC1007. We do not explain details but learn how to use the routines.

Line 1: Branching to the routine "opendos".

Line 3: Moves the constant value of 40 quickly into D0. This represents the maximum length, which will be read from the keyboard.

Line 4: Loads the effective address of the buffer "input" into A0.

Line 5: Branching to the routine "readchar".

Line 7: Adds the constant value 7 quickly to D0. If you study the two buffers, output and "input" on the program line 12-16, you will see that the text "Hello," contains 7 characters. When we add the 7 to D0, D0 will contain the record length of the "hello"-message - plus the length of the text that was loaded.

Line 8: Loads the effective address of the "output" buffer to A0.

Line 9: Branching to the routine "writechar". This will result in that the message "Hello, ?????????" will be printed onto the screen.

Line 10: Ends the program and returns to the calling instance.

Line 12-13: Here is the text "Hello," declared.

Line 15-16: Here we have reserved 40 bytes "to store the text that is read from the keyboard.

Line 17: This is new! This command is very simple. It ensures that the instructions starts at an even address. If you try to put an MC68000 instruction to an odd address (eg. address$10001) the assembler will give an error message. The reason for this is that the MC68000 has a 16 bit address-BUS (two bytes).

Line 20-32: Here is the start of the routine "readchar".

Line 34-46: Here is the start of the routine "writechar".

Line 48-62: Here is the start of the routine "opendos".

Now we are ready to try to execute this program example. To run this program from K-Seka will be useless. Boot therefore from the course-disk (disk 1) and type:

1> MC1007

Enter your name and press RETURN. You will now see that the AMIGA "says hello" to you.

You have now learned how to read and write to the CLI window. Remember that the key is to use subroutines rather than to understand each instruction.

MACHINE CODE IX

In this machine code chapter, we examine two instructions labeled EXG and CMPM.

We start with the simpler instruction first: the EXG (which stands for EXCHANGE). This instruction exchanged the values of two registers. Notice that it always changes all 32 bits. It can change both the data register to data register, data register to address register address register to address register and address register to data register.

Here are some examples:

```
EXG D0, D1
EXG D0, A6
EXG A0, A2
```

This instruction should be easy enough to understand.

Let us turn to the second instruction CMPM, which is a variant of the CMP instruction. CMPM is an abbreviation for COMPARE MEMORY.

We demonstrate this with a small program example:

```
        move.l      #$10000, a0
        move.l      #$20000, A1
loop:
        cmpm.w      (a0)+, (a1)+
        beq.s loop
        rts
```

This program example will compare the values in respectively address $10,000 and $20000. If these values are equal, it will jump up again and compare the values at address $10002 and $20002, etc. When it finds two different values it breaks the loop and returns from the program.

You can try different numbers or exchange the BEQ instruction with a BNE instruction. Try it yourself - do not forget that it is very important that you try yourself. You learn a lot from your own failure.

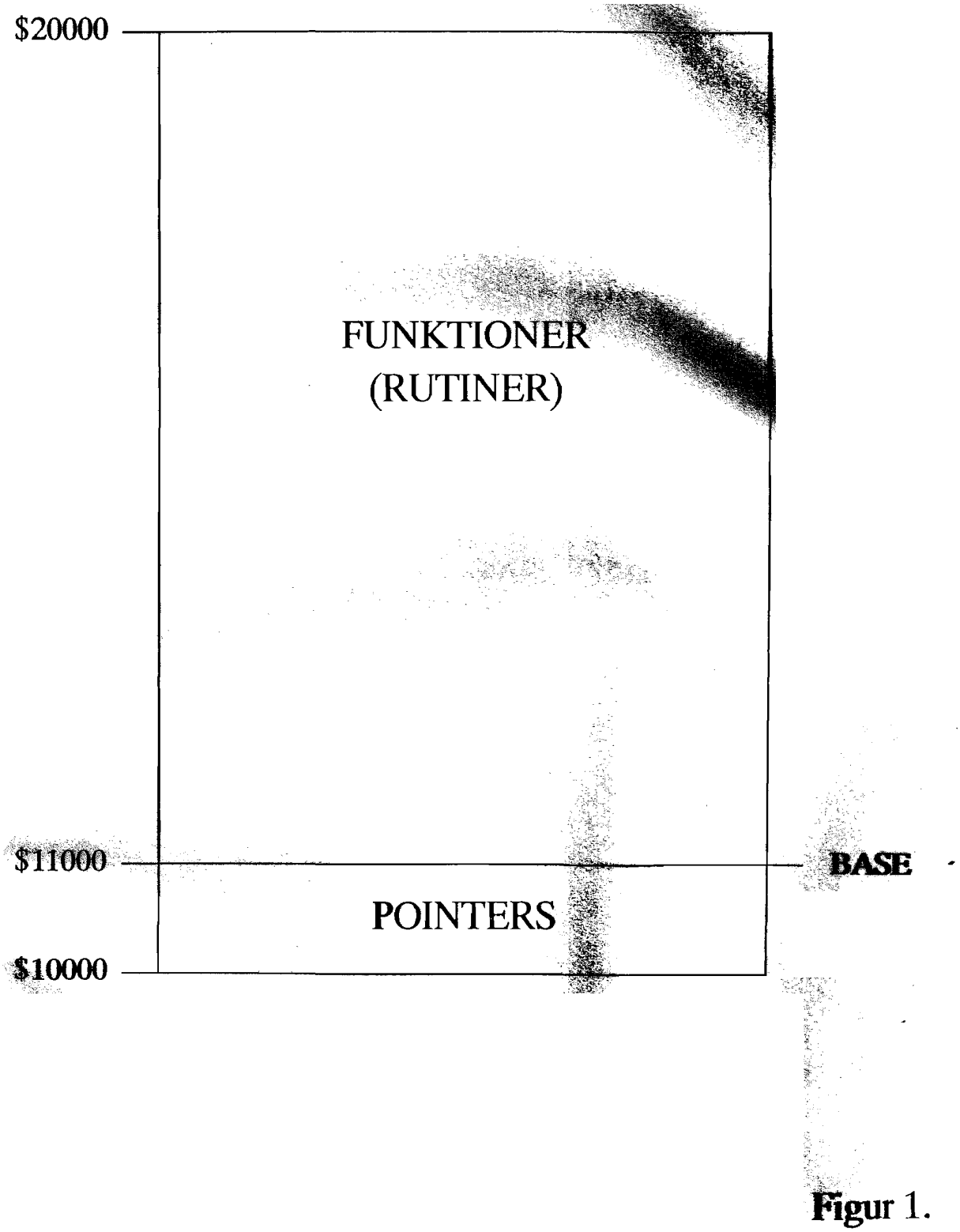
SOLUTIONS FOR TASKS IN ISSUE IX

Task 0901: The advantage of the INTERRUPT is that the processor (MC68000) can interrupt e.g. to check whether you type the keyboard and therefore can handle more routines, programs or tasks.

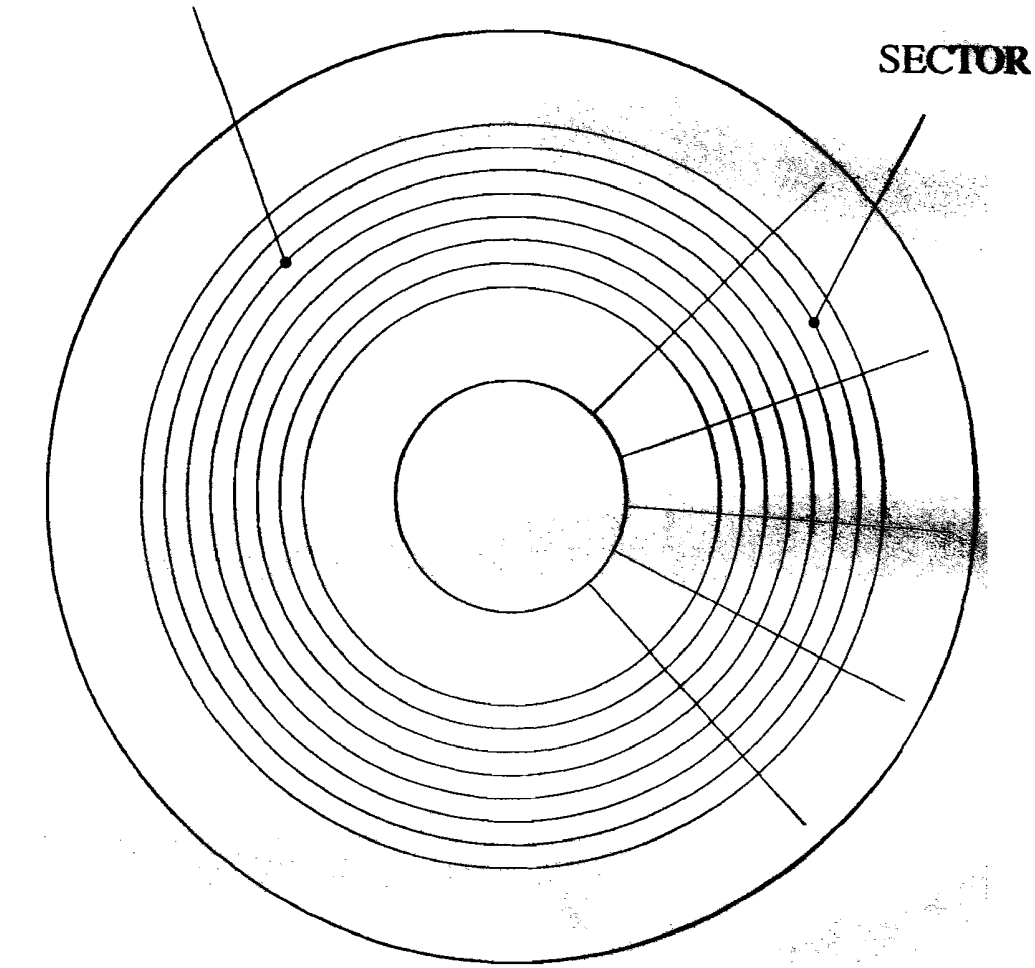
Task 0902: Interrupts 7 is often called the NONMASKABLE INTERRUPT.

Task 0903: The BCHG instruction inverts a single bit in a register or in memory (changing from 0 to 1 or vice versa).

Task 0904: The register D1 would contain: $6C9D4651



Figur 1.



Figur 2.