Programming in machine code on the Amiga

A. Forness & N. A. Holten
Copyright 1989 Arcus
Copyright 1989 DATA SCHOOL

Issue 11

DATA SCHOOL
Postbox 62
Nordengen 18
2980 Kokkedal

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

CIA CHIPS

In this issue we will look at the I/O ports of the AMIGA. There are two chips inside the Amiga, which helps to manage the parallel port, serial port, joystick ports, etc. These chips are called CIA chips.
The abbreviation CIA does not stands for "the American Intelligence "-"Central Intelligence Agency"but for Complex Interface Adapter. The two CIA chips are called the CIA-A and CIA-B.

Let us now proceed with an overview of the addresses of these chips. Note that these addresses can only addressable by byte-length (e.g. MOVE.B).

CIA-A:

ADDRESS	NAME	FUNCTION
$BFE001	PRA	Data Register A
$BFE101	PRB	Data Register B
$BFE201	DDRA	Data Direction Register A
$BFE301	DDRB	Data Direction Register B
$BFE401	TALO	HOURS A LOW-register
$BFE501	TAHI	HOURS A HIGH-register
$BFE601	TBLO	TIMER B LOW register
$BFE701	TBHI	TIMER B High Register
$BFE801		Vertical-sync-counter (BIT 0-7)
$BFE901		Vertical-sync-counter (BIT 8-15)
$BFEA01		Vertical-sync-counter (BIT 16-23)
$BFEB01		Not used
$BFEC01	SDR	Serial Data Register
$BFED01	ICR	interrupt control register
$BFEE01	CRA	Control Register A
$BFEF01	CRB	Control Register B

CIA B:

ADDRESS	NAME	FUNCTION
$BFD000	PRA	Data Register A
$BFD100	PRB	Data Register B
$BFD200	DDRA	Data Direction Register A
$BFD300	DDRB	Data Direction Register B
$BFD400	TALO	HOURS A LOW-register
$BFD500	TAHI	HOURS A HIGH-register
$BFD600	TBLO	TIMER B LOW register
$BFD700	THBI	TIMER B High Register
$BFD800		Horizontal-sync-counter (BIT 0-7)
$BFD900		Horizontal-sync-counter (BIT 8-15)
$BFDA00		Horizontal-sync-counter (BIT 16-23)
$BFDB00		Not used
$BFDC00		SDR Serial Data Register
$BFDD00		ICR INTERRUPTER control register
$BFDE00		CRA Control Register A
$BFDF00		CRB Control Register B

This was the setup for both the CIA chips. We will not review all the features in this issue. Let's explain the first four registers in each CIA-CHIP, i.e. PRA, PRB, DDRA and DDRB.

Each of these Chips is physically equipped with 8-bit parallel "port" which can be set to be either input or output. With the registers DDRA and DDRB one can determine if the ports are inputs or outputs. The registers PRA and PRB are used to either read from or write to the ports PA or PB. Note that is called "Digital Hub" which means that they can only have two different values, either 0 or 1.

Here comes the configuration of PRA and PRB-registers in both CIA chips (as they work in the AMIGA).

CIA-A PRA ($BFE001):

BIT	FUNCTION
7	Fire-button joystick port 1
6	Fire-button joystick port 0 (left moose button)
5	RDY (disk)
4	TKO (disk)
3	WPRO (disk)
2	CHNG (disk)
1	PWD (POWER led)
0	OVP (system)

CIA-A PRB ($ BFE101):

BIT	FUNCTION
0-7	Parallel Port Data 0-7

CIA-B PRA ($BFD000):

BIT	FUNCTION
7	DTR (serial port)
6	RTS (serial Port)
5	CD (serial port)
4	CTS (serial port)
3	DSR (serial port)
2	SEL (parallel port)
1	Pout (parallel port)
0	BUSY (parallel port)

CIA-B PRB ($BFDIOO):

BIT	FUNCTION
7	MTR (disk)
6	SEL3 (disk)
5	SEL2 (disk)
4	SEL1 (disk)
3	SEL0 (disk)
2	PAGE (disk)
1	DIR (disk)
0	STEP (disk)

We have now explained the addresses of the various registers in the chip and took a closer look at the configuration of the PRA and PRB register.

In the next chapter, we continue with the explanation of program example MC11O1.

READING OF MOUSE POSITION

This chapter outlines the program example MC1101, namely reading the mouse position. When we read the mouse position (the X and Y coordinates) register $DFF00A is used. This register contains two 8-BITS values. BIT 0-7 provides the X-position, while BIT 8-15 contains the Y position. Each byte may contain values from 0 to 255 (8 BITS).
When you move the mouse to the right, the X-position will increase. If you move the mouse to the left, the X-position decrease.

The disadvantage of using these values directly is that when mouse moved so far that the X position is over 255 the register will wrap (overflow) and start over from 0 again. We therefore create a routine that calculates the distance between the new position and the previous one.

Study the examples below:

OLD POSITION	NEW POSITION	OFFSET
130	150	+20
100	75	-25
250	10	+16
50	255	-51
180	180	0
0	255	-1
100	200	+100
100	250	-106

The calculation is done like the following:

Let us say that "new position" is 250, and "OLD POSITION 'is 100. First subtract the old position from the new.

250 to 100 = 150

If the result is greater than 127 ➔ subtract 256 from it.

If the result is less than -128 ➔ set it 256

In this case the value exceeds 127, and we must therefore subtract 256 from it.

150-256 = -106

So: The result is -106.

Let us take some examples:

NEW POSITION = 22
OLD POSITION = 200

22 - 200 = -178

value is less than -128 ....

-178 + 256 = 78

Offset = +78

NEW POSITION = 160
OLD POSITION = 200

160-200 = -40

value is either too small or too big ....

Offset = -40

It should be clear right now, isn't it?

Now we explain the program example MC1101:

Line 1:	This is the start of the main routine.
Line 2:	Jumps to routine "mouse".
Line 4:	Loads the effective address of "mouse-x" into A1.
Line 5:	Loads the effective address of "mouse-y" into A2.
Line 7-8:	Moving the X position into D1, and Y-position into D2. The values are not used for anything anywhere in this program example, but you know how to create a program that controls a sprite using the mouse.
Line 12-13:	Checks if the left mouse button is pressed. If not jump back to the "main".
Line 17:	Here begins the mouse routine.
Line 18:	Stores all registers onto the stack.
Line 19:	Moves the value located at address $DFF00A into D0.
Line 20:	Masks out not used bits only the first 8 bits (BIT 0-7) are kept in D0.
Line 21:	Moves 0 into D2 (clears it). This value is the smallest X position we want.

Line 22: moves the value 639 into D3. This value indicates the maximum X position. We have now set the field for X-position to have values between 0 and 639
Line 23: Loads the effective address of "oldx" into A1.

Line 24: Loads the effective address of "mousex" into A2.

Line 25: Branch to the sub-routine "calcmouse".

Line 26-33: This code is for the Y- position – it is similar to the code for the X-position. We have now set the field for Y-position to have values between 0 and 511.

Line 34: Restores the old register values back from the stack.

Line 35: Return from subroutine.

Line 36: Here begins the routine that calculates the displacement and check that it is within the defined area (X = 0 - 639, and Y = 0 - 541).

Line 37: Moves the constant value 0 into D1.

Line 38: Moves the value in "oldx/oldy" into D1 depending from which part of the “mouse” routine the call came.
Line 39: Moves the value from D0 (the current value) into "oldx/oldy".

Line 40: Moves the value from D0 (the current value) into D5.

Line 41: Moves the value from D1 (oldx/oldy) into D6.

Line 42: Subtracts the value in D0 (new position) from the value in D1 (old position). You subtracted the new value of the position from the old – the result is in D1.
Line 43: The result of the subtraction in D1 is compared with -128.

Line 44: If the value in D1 is less than -128, jump to "mc_less".

Line 45: If not let than -128 the result in D1 is compared with 127.

Line 46: If the value in D1 is greater than 127, jump to "mc_more".

Line 47: Is the resulting difference 0 (compare the result with 0)?

Line 48: If the result in D1 is less than 0, branch to "mc_chk2".

Line 49: Continue here if D1 is greater than (or equal to) 0.

Line 50: Compares the value in D6 to the value in D5.

Line 51: If D6 is greater or equal than D5, jump to "mc_chk1ok".
Line 52: Swapping sign of the value in D1. So: If D1 was 100 it becomes -100, and vice versa.

Line 54: Jumps to "mc_storem".

Line 56: Compares the value in D6 to the value in D5.

Line 57: If D6 is less or equal than D5, jump to "mc_chk2ok".

Line 58: Swapping the sign of the value in D1.

Line 60: Jumps to "mc_storem".

Line 62: Adding the conatant value of 256 to the value in D1.

Line 63: Jumps to "mc_storem".

Line 65: Subtracts the constant value of 256 from the value in D1.

Line 67: Swapping the sign of the value in D1.

Line 68: Adds the value in D1 to the "mousex/mousey."

Line 69: Moves the "mousex/mousey" value into D0.

Line 70: Compares the value in D0 to the the value in D2.

Line 71: If the value in D0 is less than the value in D2, branch to the "mc_tosmall". So: that's the case if you try to move mouse outside the defined area.

Line 72: Compares the value in D0 to the value in D3.

Line 73: If the value in D0 is larger than the value in D3, brach to "mc_toolarge". This happens if you move the mouse outside the defined area.

Line 74: Ends the routine ("calcmouse").

Line 76: Moves the value in D2 into (A2). If the value is too small to be within the area, the least legal value is moved into the "mousex/mousey."

Line 77: Exiting routine.

Line 79: Moves the highest legal value directly into "mousex/mousey."

Line 80: Exiting routine.

Line 81-88 Here are the variables declared.

In the next chapter, we look at the Amiga printer port.

PARALLEL PORT

In this section we look at the printer port (parallel port) of the Amiga.

The parallel port is an interface that can be used to manage a printer. The parallel interface transfers data in parallel. This means that 8 bits (one byte) are transferred at a time (unlike the serial port transfer which only can transfer one BIT at a time). Technically the parallel port works as follows:

- First the Amiga sends a BYTE to printer.

- Then the Amiga is waiting for the printer that it processes the byte. When printer has processed the byte, it will send a "clear" signal back to the Amiga. When the Amiga has received the "clear" signal it may send another byte to the printer.

The printer also has the opportunity to send other messages to the Amiga. In addition to the "clear" signal (READY), the printer can send messages that it is out of paper or the printer is not "ONLINE". We can also check whether there are any printers connected, or if the printer is not turned on.

.... and now let’s have a look at the program example MC1102:

Line 1:	Loads the effective the address of "buffer" into A0. The "buffer" contains the data (characters) which is to be sent to the printer.
Line 3:	Turn off all interrupts. This must be done To avoid problems with the operating system (AmigaDOS). If you omit this step, you will get a "PRINTER Trouble" message on screen after you have used the printer for a short time.
Line 4:	Branches to the routine "print". This routine performs the actual printing.
Line 5:	Turns on all interrupts again.
Line 7:	Exits the program.
Line 9-11:	Here are the characters defined which are to be printed. The value 10 at the end of the sentence indicates a line break (LINE FEED, LF) value of 0 are always at the end of the text so that "print" -routine will end at this byte.
Line 14:	Here begins "print"-routine.
Line 15:	Moves the constant value of $FF into $BFE301 (= DDRB the data direction register for Port B of CIA-A). This ensures all lines of PRB-port are switched to output-mode (since data is sent to the printer).
Line 18:	Moves a byte from the address $BFD000 (Portregister A from CIA-B) into D0. To Port-A of CIA-B the printer sends the signals to the Amiga.

Line 19:	Performs a logical AND. This masks out the bits 7-3 and keeps only bit 0-2. (#% 111 is the binary form of= # 7).
Line 20:	Compares the value in D0 with the binary value %100 (= 4 decimal).
Line 21:	If the value in D0 is equal to %100, branch to "ready" which means the printer is ready to accept data.
Line 22:	Compares the value in D0 with %001.
Line 23:	If the value in D0 is equal to %001, branch to "offline" which means that the printer's "ONLINE" button is off ;-)
Line 24:	Compares the value in D0 with %111
Line 25:	If the value in D0 is equal to %111, branch to "poweroff" which means the printer is not turned on. Notice that the same happens if no printer is connected to the Amiga.
Line 26:	Compares the value in D0 with %001
Line 27:	If the value in D0 is equal to %001, branch to "wait".
Line 28:	Compares the value in D0 with %011.
Line 29:	If the value in D0 is equal to %011, branch to "paperout" which means if the printer runs out of paper, it will be branched to "paperout".
Line 30:	Branch up again to "wait".
Line 32:	Here begins the routine that sends a character from the buffer (the address in a0) to the printer.
Line 33:	Moves a byte (character) from the buffer (the bruffers address is in A0) into D0, and and increase the address of the buffer by one. This means one character is copied into D0.
Line 34:	Compares the value in D0 to 0
Line 35:	If the value in D0 is equal to 0, jumping to "stop". This means that the end of the buffer has been reached and the text is finished.
Line 36:	Moves the value in D0 into the address $BFE101 (Port Register B of CIA-A). This makes sure that we send a character to the printer.
Line 37:	Branch back to "wait".
Line 39:	Start of the routine when printing is finished.

Line 40-41:	Moves the constant value 0 into D0. This is done because you must be able to check whether an error has occurred during printing. So: when the "print"-routine returns 0 in D0 there were no printing errors. End the routine with RTS.
Line 43-45:	When the printer is turned off, (or no printer is connected) the constant value of 1 is moved into D0 and the routine is ended with RTS.
Line 47-49:	When the is not "ONLINE" (i.e "OFFLINE" ;-) the constant value of 2 is moved into D0 and the routine is ended with RTS.
Line 51-53:	When the printer ran out of paper the constant value of 3 is moved into D0 and the routine is ended with RTS.

The function of this printer routine can be summarized as follows:

IN:	A0 = address of the text to be printed.	
OUT:	D0 = status:	0 = ok
		1 = poweroff (not connected)
		2 = offline
		3 = paperout

Program example MC1102 communicates directly with the printer. It can be both an advantage and a disadvantage. The advantage is that it goes quickly, simply and seamlessly. The disadvantage is that if you make a program which uses control codes this program does not work with all types of printers. Control codes are used to manage thins as **bold**, <u>underline</u> and font size, etc.

As you probably know, the Amiga has a “PREFERENCES” program where you can specify which type of printer you have. It has no influence on this program example. To use Amiga printer routines you have to use a different method. It can be done using the "WriteFile" method in chapter 10. The filename is set to "prt", and the buffer must contain the text to be sent. If you have a printer, you can try this as a small experiment. In the next section, we continue with MACHINE CODE X, where we will discuss among other topics how to set PIXELs on a graphics screen in detail.

MACHINE CODE X

This section of machine contains some programming tips. The first thing we should look at is a routine that sets a pixel at a graphics display. X- and Y-position are stored in D0 and D1. Here comes the routine:

```
1       pixel:
2               mulu        #40, D1
3               move.w      D0, D2
4               lsr.w       #3, D0
5               not.b       D2
6               andi.w      #7, D2
7               add.w       D1, D0
8               lea.l       screen, A1
9               bset        D2, (A1, D0.w)
10              rts
```

Here comes then the explanation:

Line 2:	D1 (Y-position) is multiplied with the 40 – this because we assume that the screen width is 320 pixels (320 / 8 = 40).
Line 3	Move the value in D0 (x-position) into D2.
Line 4:	Shift the bits in D0 to the right by three. This is the same as dividing by eight so we get the byte position for the x-position.
Line 5:	Inverts the bits of the lowest byte in D2
Line 6:	Mask with #%111 (=7 dec) thus leaving bit 0-2 intact in D2. This register now contains the bit position.
Line 7:	Adds the value in D1to the value in D0 this is the complete offset in bytes. Register D0 contains now the byte position on screen.
Line 8:	Loads the effective address of the "screen" into A1.

Line 9: The value in D indicates the BIT which is to be set at the address of the scree plus its offset in bytes (A1 + D0). This will results in a set pixel on the screen at the x- and y-position (which were passed in D0 and D1).

Line 10: Exiting routine.

You probably know that different instructions have different execution times. There are especially two instructions that take a very long time - namely multiplication (MULU or MULS) and division (DIVU or DIVS). Mulu uses approx. 15 to 20 times longer than, for example "MOVE.L D0, D1", whereas DIVU needs up to 45 times longer than "MOVE.L D0, D1".

In the program example above, we used a MULU. Once you are more skilled in programming you should test your routines for speed and avoid MULU, MULS, DIVU and DIVS if possible. To perform a multiplication by 40, shifting bitwise is possible. If we want to multiply D1 for instance by 32 this would be indeed very simple (lsl.w #5, D1). The number 40 cannot be as easily achieved by shifting as the well known "binary" numbers 32 (1, 2, 4, 8, 16, 32, 64, 128 etc.). We will nevertheless show you how achieve a multiplication with bit shifting in the following program example (number to be multiplied is in D0):

```
1       move.l          D0, D1
2       lsl.w           #5, D0
3       lsl.w           #3, D1
4       add.w           Dl, D0
```

Line 1: Moves (copies) the value in D0 into D1.

Line 2: Shifts the content of the D0 five bits to the left. This means that the content is multiplied by 32. Note that this is only applied to the lower 16 Bit – if you want to shift the whole 32Bit of the register you must use “lsl.l”.

Line 3: Shifts the content of D1 three bits to the left. This means that the content is multiplied by 8.

Line 4: Adds the value in D1 to the value in D0.

Line 5: Exits the program.

Let us show some examples of how it works:

```
D0    =     1
1     *     32    =     32
1     *     8     =     8
32    +     8     =     40      (1 * 40 = 40 ;-)
```

We try this one...

```
D0    =    3
3     *    32    =    96
3     *    8     =    24
96    +    24    =    120    (3 * 40 = 120)
```

and one for ...

```
D0    =    25
25    *    32    =    800
25    *    8     =    200
800   +    200   =    1000   (25 * 40 = 1000)
```

It perfectly works! We now have avoided a MULU instruction. If you want to be even more advanced, keep in minf that using “LSL.W #5, D0” takes longer than “LSL.W #3, D0”. So: The more bits you shift, the longer it takes to execute the instruction. We can even save more time if we rewrite the previous program.

```
1    lsl.w     #3, D0
2    move.l    D0, D1
3    lsl.w     #2, D0
4    add.w     D1 D0
```

Line 1: Shifts the value in D0 to the left by three bits. It’s an multiplication by 8.

Line 2: Moves (copies) the value in D0 into D1.
Line 3: Shifts the value in D0 to the left so that the value in D0 is now shifted five times in total.
Line 4: Adds the value located in D1 to D0.

In the first example the program performed eight shifts (5 +3) while in this program example only five (3 +2) shifts were performed.

So, our routine has become again faster. We now show the complete "Pixel"-Set routine with our new multiplication method:

```
pixel:
lsl.w     #3, D1
move.w    D1, D3
lsl.w     #2, D1
add.w     D0, D1
move.w    D0, D2
lsr.w     #3, D0
not.b     D2
andi.w    #7, D2
add.w     D1,D0
lea.l     screen, A1
bset      D2, (A1,D0.w)
rts
```

This routine you can use in your own programs. The benefit of this routine is that you can set about 50 000 pixels in a second. Speeding up this routines is difficult. The only thing you can do is to move program line 11 outside the routine. If you call the routine many times in arrow, there's no reason to get screen address every time.

As a summary, this routine uses the following in/out parameters:

IN:	D0	=	X-position
	D1	=	Y-position
OUT:	Nothing		

The next we will look a is how to read the joystick position. In our routine to read out the mouse position we used the register $DFF00A. This register belongs to Joystick port 1. Since on this port 1 the mouse is plugged in most of the time, our routine deals with the joystick port 2. The register for JOYSTICK port 2 is $DFF00C. Here comes the routine:

```
1       readjoy:
2       move.w          Dl, -(A7)
3       move.w          $DFF00C, D0
4       move.w          D0, D1
5       andi.w          #3, D0
6       lsr.w           #6, D1
7       andi.w          #12, D1
8       add.w           D1 D0
9       move.w          (A7) +, D1
10      rts
```

Line 1: routine called "readjoy"

Line 2: Moves (copies) the content of D1 onto the STACKS. Notice that only the low-word (bit 0-15) is stored on the stack. We do not need to store the whole longword of the register because we only use BIT 0-15 in register D1 in the program routine. This makes the routine a little bit faster and we save 2 bytes (one word) on the STACK.

Line 3: Moves the word located at address $DFF00C into D0.

Line 4:	Moves the value in D0 into D1. So: D1 is a copy of D0.
Line 5:	All bits except bit 0-1 are masked out. This is done by a logical AND such that only BIT 0 and 1 remain in D0 (#3 = % 0000000000000011).
Line 6:	Shifts the content of D1 to the right by six.
Line 7:	All bits except bit 2-3 are masked out. This is done by a logical AND such that BIT 2 and 3 remain in D1 (#12 = %0000000000001100).
Line 8:	Adds the value in D1 to the value in D0.
Line 9:	Retrieving the old value of D1 from the Stack.
Line 10:	Exiting routine.

As you see, BIT 0, 1, 8 and 9 are important and we need to retrieve them from the address $DFF00C. All other bits are not needed. This program results in that we have a 4 bit value of (bit 0-3) in D0.

In Figure 1 at the end of this chapter you will see what values the different directions of the joystick have. You can of course experiment with a program that manages a Sprite (or Bob = blitter object) on screen using a joystick.

COMMENTS ON CHAPTER XI

That was the penultimate chapter. We have come quite far around during the review of the different machine code instructions since we started with the different systems of numbers. Some instructions, we have carefully examined, and some we have just touched. It is always difficult to choose which topics to deal with, but we hope that you have a enough knowledge of machine code programming that you can further work on the subject on your own.

The next chapter is the last thing we will come up with different tips and tricks that are good to know when you are on your own in the large world of programming.
Together with chapter XII a disk will be released. It will be difficult to take full advantage of this chapter without having the disk at your hands. We therefore recommend you to order it together with the last chapter.

We read us in the 12th chapter.

Sincerely,
DATA SCHOOL

Carsten Nordenhof