Programming in machine code on the Amiga

A. Forness & N. A. Holten


Issue 9

Content


DATA SCHOOL
Postbox 62
Nordengen 18
2980 Kokkedal

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

**INTERRUPTS I**

We begin this chapter by explaining what interrupts are in general. We will use the term INTERRUPT because it is the one mostly used by programmers.

If computers would not offer INTERRUPTS, it would be similar to a situation like this :

You are sitting and waiting for the phone to ring, but when ringing-function of the phone does not work (it's broken), you can not hear if it rings. So you have to pick up the phone all the time and say "hello?" and if no one answers, then hang up again (this is called polling).

Maybe you sit and read a magazine – or examine the today's post. The broken ringing-function of your phone will result in that you will not read more than a few sentences, before you again have to lift the receiver of the phone to check if someone is calling. If you have multiple phones there, and must constantly check the phone, you will not read much. That's annoying, isn't it?

An example of an interrupt in a computer is, e.g., reading out the keyboard. If we could not take advantage of Interrupts, we had to check the keyboard all the time to be sure the computer got all the typed characters.

But because there is an interrupt function, your program does not have to check the keyboard at all. The reading of the keyboard is performed automatically by a so-called interrupt routine. This routine will be performed as soon as you press a key down on the keyboard. In this case the routine reads out the key-strokes into a keyboard buffer - that is accumulating the characters that you enter in a small area in memory. If a program is started to retrieve data from a diskette it can therefore finish disc access before it downloads any keystrokes that you've made in the meantime - and which are stored in the keyboard buffer.

This should be a fair introduction to the concept of INTERRUPTS. In the next chapter we will proceed with how INTERRUPTS work within your favorite machine the AMIGA.

**INTERRUPTS II**

In the Amiga there are 7 different INTERRUPTS. They are numbered from 1 to 7. The seventh INTERRUPT can not be turned off and is therefore called NONMASKABLE INTERRUPTER. The other INTERRUPTS can be turned off if needed. So if you want to listen to an interrupt from, e.g., the keyboard, so it is possible. Each INTERRUPT in the Amiga has a certain priority. Let us take a closer look at just this:

| Interrupt | function in the AMIGA |
|---|---|
| 7 | External (external) Interrupt |
| 6 | External INTERRUPTS |
| 5 | Disc |
| | Serial port |
| 4 | Audio |
| 3 | Blitter |
| | Vertical-blank |
| | Copper |
| 2 | I/O ports and TIMERS |
| 1 | is used as software interrupts |
| | Disk |
| | Serial Port |

INTERRUPT 7 can be completely ignored. This is because that in time of writing there is no equipment to use to trigger this interrupt. We concentrate therefore on the INTERRUPTS numbered from 1 to 6. INTERRUPTS are divided into priorities where INTERRUPTER 7 has the highest priority, and interrupt 1 the lowest priority. This works so that while INTERRUPTER 3 is served, an INTERRUPT with a lower or same priority can not stop/interrupt it to being served.

On the other hand, interrupts with higher priority may stop/interrupt a lower one in order to carry out its routine first and then let the lower interrupt finish its routine. This is designed because each device (disk, blitter, copper, etc.) may need a slightly faster dispatching of its routine than other units.

Also note that, for example. INTERRUPT 3 (see the table) has more features. This leads to interrupt ROUTINES that must check which device generated the INTERRUPT to get to know what interrupt ROUTINE should be performed. Later more details are explained.

Let us now go one step further and talk about how the different INTERRUPTS are "created":

INTERRUPTER 6: This interrupt is like INTERRUPTER 7, also coupled to external devices. This means that not INTERRUPT is triggered if you do not have, e.g., a hard drive etc.

INTERRUPTER 5 (DISK): This interrupt is generated (triggered) when a so-called DISC SYNC WORD found on the disk. We can explain this in all simplicity: the data on a disk is stored in rings on the disk - the so-called track. To AMIGA needs to know the beginning of the track, there is a special combination of ones and zeros (16 bits) at this location. When the AMIGA reads this combination on a floppy, it will jump to interrupt routine, to startup the disk-DMA to begin to read in the data. This should go quickly, and therefore this function has a high priority.

INTERUPT 5 (Serial Port): This interrupt is activated when the buffer to the serial port is full. The serial port is an interface that can send or receive data from other entities, e.g., a MIDI device, a modem, a second computer - or similar. The special feature of serial transmission is that only one bit at a time is transferred. When the serial port has received 8 bits (one byte) to a buffer register it is emptied immediately before the next BIT coming in, so it does not "run over" – which is called OVERFLOW. So when the buffer register is full, this will trigger this interrupt and jumping to its interrupt routine, which places the value from the buffer register to a place in RAM where it can be safely accessed. As you see also this function has a high priority to make sure that the data on the serial port will be received correctly.

INTERRUPTER 4: This interrupt is generated when an audio channel has completed to play a SAMPLE. This means that we can switch off the interrupt routine so the audio sample is only played once through the DMA, instead of letting the sample play over and over again. Notice that all four audio channels are "switched" on this interrupt, so you must find out what audio channel, has come to the end of the sample.

INTERRUPTER 3 (Blitter): This interrupt is generated when a blitter operation is finished. In our program examples we use these (now famous) lines to check this:

```
wait:
        BTST        #6, $DFF002
        BNE wait
```

The second method is to create an interrupt routine, for example which sets up and starts the next blitt.

As you see this interrupt has not a high priority. This comes because of the fact that this operation does not need one so quick attention (or care) as the serial port in interrupt level 5.

INTTERUPT 3 (vertical-blank): This interrupt is generated every time the electron beam jumps up to the line 0 so, this INTERRUPTER is performed each time the screen is updated. It has no special feature in itself, but the interrupt routine may, for example update the position of the WORKBENCH arrow and other similar things (text scrolling) for each screen update. We could for example create an interrupt routine that updates a scroll-text.

The main program would look like this (the code is moved to the interrupt routine):

```
main:
        BTST  #6,$BFE001
        BNE main
```

instead of .....

```
main:
        MOVE.L      $DFF004, D0
        ASR.L #8, D0
        ANDI.L #$1FF, D0
        CMP.W #0, D0
        BNE main
        BSR scroll
        BTST #6, $BFE001
        BNE main
```

INTERRUPT 3 (COPPER): This interrupt can be triggered by the COPPER. For example you can let COPPER wait until line 200, and then trigger an interrupt. In practice this interrupt in the same case as the above (vertical blank) interrupt, but with the difference that you can decide at which position of the display the interrupt is triggered (i.e. not fixed to the line 0 as the vertical-blank interrupt). You can also make more interrupts for each screen update if you wish.

INTERRUPT 2: This interrupt is generated by an i/o unit controller like the parallel port, the mouse, the keys and some of the disc functions etc. We will come back with an explanation of this interrupt in issue 11 when we shall discuss this I/O unit.

INTERRUPT 1 (software): This interrupt is rather unusual. It is driven by the running program. As you will understand that this interrupt has no special function, but we can use this feature for multitasking. This means - as you might already know - that two or more tasks (programs) can run parallel. This is not entirely true, because what actually happens is that the processor switches between the programs – the running an the interrupted program all the time.

Or, when a program has been carried out for a little while (time may vary), so the program will even trigger this INTERRUPT itself. The interrupt routine then provide code for switching the context to the next program. Thus, this interrupt routine distributes "processor power" over the different running programs.

INTERRUPTER 1 (DISC): This interrupt is generated when the disc-DMA has finished loading data from a floppy. This interrupt routine can for example move the R/W-head to start up a new load. It has a substantial lower priority than the disk function in interrupt 5 and it is designed so because this function does not need such a quick treatment.

INTERRUPTER 1 (SERIAL PORT): This interrupt is generated when the buffer to the serial port is empty. So: When the machine sent 8 bits (Note that serial port can be configured to send/receive 9 BITS at a time too) an interrupt routine is called, which can initialize the next byte to be sent out using the serial port.

We have now examined the functions of the individual in interrupts in the AMIGA. In the following chapter we shall look at how to create an own interrupt routine and other things.

**INTERRUPTER III**

We start this chapter with a table of contents in memory addresses $000000 - $0003FF.

| ADDRESS | DESCRIPTION |
|---|---|
| $000000 | reset, SSP |
| $000004 | reset, PC |
| $000008 | bus error |
| $00000C | address error |
| $000010 | illegal instruction |
| $000014 | division by zero |
| $000018 | CHK instruction |
| $00001C | TRAPV instruction |
| $000020 | privilege violation |
| $000024 | trace exception |
| $000028 | unimplemented instruction (1010) |
| $00002C | unimplemented instruction (1111) |
| $000030 | not used |

| ADDRESS | DESCRIPTION |
|---|---|
| … | |
| $000060 | spurious interrupt |
| $000064 | interrupt 1 auto-vector |
| $000068 | interrupt 2 auto-vector |
| $00006C | interrupt 3 auto-vector |
| $000070 | interrupt 4 auto-vector |
| $000074 | interrupt 5 auto-vector |
| $000078 | interrupt 6 auto-vector |
| $00007C | interrupt 7 auto-vector |
| $000080 | TRAP #0 |
| $000084 | TRAP #1 |
| $000088 | TRAP #2 |
| $00008C | TRAP #3 |
| $000090 | TRAP #4 |
| $000094 | TRAP #5 |
| $000098 | TRAP #6 |
| $00009C | TRAP #7 |
| $0000A0 | TRAP #8 |
| $0000A4 | TRAP #9 |
| $0000A8 | TRAP #10 |
| $0000AC | TRAP #11 |
| $0000B0 | TRAP #12 |
| $0000B4 | TRAP #13 |
| $0000B8 | TRAP #14 |
| $0000BC | TRAP #15 |
| $0000C0 | Not used |
| ... | |
| $000100 | user interrupter vector |
| ... | |
| $0003FC | user interrupter vector |

We concentrate on the addresses $64 to $7C. These memory areas contain pointers (vectors) to the different programming routines that are executed when an interrupt is triggered. So: when the processor (MC68000) gets a signal of type interrupt 1, it will retrieve the value that is on address $64 (long word), and then treat this value as an address to which to jump to. The processor also makes a few other things before jumping to an interrupt routine. Firstly it does moves the value located in the program counter (PC) to the STACK. Then store the STATUS register to the STACK, and finally jumps to the interrupt routine. So: It has to store the program counter so that it knows where to continue when the interrupt routine is finished (this is the same as the BSR). Then the STATUS register is restored (the register contains, e.g., ZERO flag Carry-f1aget, etc.).

We illustrate it further with an example:

```
CMP.W #25, D0
BEQ loop
```

As we already know An interrupt can occur at any time fully independent of other programs. Imagine that a interrupt occurred between two instructions in the example above. If D0 ,e.g., was 25, this CMP instruction sets the ZERO flag to 1 When the interrupt then is triggered before the BSR instruction the ZERO flag may have a different value (because of the interrupt routine) before the processor jumps back to go at BEQ instruction. This may lead to that the BEQ instruction will branch "wrong" a few times. Therefore it is necessary that the processor also stores the STATUS register.

All this is done automatic, so that you do not need to think about this when you are programming an interrupt routine. However, there are other things you should think about, namely the data registers and address registers. These are not stored automatically during an interrupt. Therefore, the first thing if you create an interrupt-routine would be to store the registers that you need to the STACK if you must use, e.g., the data registers D0 to D3 and the address registers A0 to A1 in your routine. To store these registers on the STACK look at the following code (see also the machine code chapter where we explain this variant of the MOVE):

```
MOVEM.L     D0-D3/A0-A1, -(A7)
```

When the interrupt routine is completed, you finally restore the registers with the following instruction in order to get the values back into the registers.

```
MOVEM.L     (A7)+, D0-D3/A0-A1
```

The instruction used to terminate a routine is usually an RTS. But what makes this RTS instruction anyway? Yes, it retrieves a value from Stack (A LONG WORD), and uses this value to jump back to who it came from. However, we already told you that when an interrupt occurs not only the program counter (PC) but also the status register are stored on the stack. Therefore we can not use RTS to end an interrupt routine. For that reason we have an instruction called RTE (return from exception). This instruction will also ensure that the status register is restored. Therefore, the structure of an interrupt routine looks like this:

```
Interrupt:
        MOVEM.L D0-D7/A0-A6, -/A7)

        ..... (some meaningful code)

        MOVEM.L (A7) +, D0-D7/A0-A6
        RTE
```

We are now reviewing the program example MC0901, located on the course disk 1.

Line 1: loads the effective address of the "jump" into A1. Notice that the label "jump" points to an instruction (program line 42)

Line 2: moves the value, located at address $68, to the address A1 +2 are pointing to. This instruction moves a value into the instruction at program line 42:
First we get the value which is located at address $68. This address contains a pointer (vector) to the INTERRUPT level-2 routine - which will be stored to the JMP instruction at program line 42. The JMP instruction performs a jump like the BRA. The difference between these two is that BRA uses an offset from the current instruction location, while JMP jumps directly to a given address (absolute address – see also the machine code chapter later in this issue). As you look at line 42, this JMP instruction is initially set to address $0. When we have done this MOVE instruction the JMP instruction will have a new address - the address which points to the old interrupt routine.

Line 4: Moves quickly the constant value of 100 to register D0.

Line 6: Moves quickly the constant value of 1 to register D1.

Line 7: Swapping the Word in D1. This leads to D1 containing $10000

Line 8: Moves the value from address $4 into the A6.

Line 9: This is a new instruction. JSR means jump to sub routine, or jump to (see also machine code chapter in this issue). In a issue X, we will work in detail on JSR commands. In all simplicity what lines 4 to 9 do is to allocate memory of 100 bytes. This means that we ask the operating system to reserve a memory block of 100 Bytes for us. Doing so we obtained a place in memory where the interrupt routine can be safe. Do not worry so much about it right now - it's important to understand how interrupt routines work, but to know where in memory it is located.

Line 11: Moves the value of D0 into A1. The routine (system routine for memory allocation) which was performed by program line 9, returns the address of the reserved memory block in d0. This address points to the first byte of the 100 byte block which was allocated.

Line 12: Moves the value in D0 into D7.

Line 14: Loads the effective address of "interrupt" into A0.

Line 15: Moves the constant value of 24 quickly into D0. It is used as a counter.

Line 18: Copies the long value, A0 points to, to the address A1 points to. Then both both addresses (in a0 and a1) are increased by 4 bytes or 1 long word.

Line 19: Subtracts the value 1 from D0 and check if D0 is negative - if not branching back to the label "copy loop". After finishing this loop the "interrupt"-routine (program line 28 to 42) was copied into the "block" of memory that we allocated earlier.

Line 21: Turns off all interrupts.

Line 22: Moves the value which lies in D7 in to address $68. The address of the memory block that we allocated earlier and we copied our interrupt routine to, is put into the pointer for interrupt level 2. Notice that we switched off all interrupts in the previous line. This must be done before you submit a new value into the interrupt vectors.

Line 23: Turns on all interrupts again. Now our interrupt routine is called when there is a signal for an interrupt level 2. When our interrupt routine is executed, it will not return to the main program - but through the JMP instruction at line 42 - jump to the old interrupt routine (which belongs to the operating system). This allows us to keep the operating system intact. So we have just "sneaked" our own interrupt routine before entering the operating system's own interrupt routine. This is known to link or hook into a routine.

Line 25: Ends the program (NOTE: Interrupt routine does not finish).

Line 27: Here is our custom interrupt routine.

Line 28: Stores the value in D0 onto the stack. We do not need a MOVEM instruction here, because we only need to store a single register. MOVEM is used only when we want to store more registers at a time.

Line 29: Moves the byte value located at address $BFEC01 into D0. This address contains the codes to be sent from the keyboard when you press a key. We will explain this register in more detail in issue XI.

Line 30: Inverts the byte in register D0.

Line 31: Rotating the byte in D0 by 1 bit to the right. More detailed explanation of this instruction can be read in the machine code chapter.

Line 33: D0 is compared with the constant value of 59.

Line 34: If D0 was not 59, jump to the label "wrongkey". Program lines 29 to 34 reads the keyboard and checking whether a special key has been pressed.

Line 36: This instruction does a bit change. It inverts bit 1 at address $BFE001. Bit 1 of this address controls the "POWER" led which respectively is switched on and off. This instruction will therefore result in a change of the power-led. This instruction is also explained in machine code chapter.

Line 39: Restoring the old contents of D0 from the stack.

Line 42: Jumps to the old interrupt routine.

To run this program, you must assemble it as usual. Then all you need is the command "j" to start it. After you run the program, try pressing "F10" while looking at the "POWER" lamp.

That was the chapter on INTERRUPTS. We may mention that in issue XII some slightly more complicated INTERRUPT routines are explained (for MIDI) which will take care of automatically sending and receiving data on the serial port.

**MACHINE CODE VIII**

In this chapter we review the following instructions: MOVEM, JMP, JSR, BCHG, RTE, ROR and ROL.

Let's start with the MOVEM instruction. The additional letter "M" in this instruction means multiple. So: This MOVE moves several things at once and is often used to store register content to and retrieve it back from the STACK:

This code …

```
MOVEM.L     D0-D2, -(A7)
```

…does the same as these 3 lines:

```
MOVE.L D0, -(A7)
MOVE.L Dl, -(A7)
MOVE.L D2, -(A7)
```

Another example:

```
MOVEM.L (A7)+, D1/A0-A4
```

The first example stores data registers D0, D1 and D2 at to the stack. The second example which performs the same as the first but it needs both more space and more time to execute.

The third example retrieves D1, A0, A1, A2, A3 and A4 from the stack. You can also use other destination address than that in the A7 to store and retrieve data. In this way you can make your own stack. Such a program could, for example. look like this:

```
LEA.L       mystack, A0
MOVEM.L     D0-D5/A1-A4, -(A0)

...
MOVEM.L     (A0)+, D0-D5/A1-A4
...
BLK.B       100,0
mystack:
```

The next instruction we must look at the JMP. This instruction
means JUMP. The difference of this instruction and the BRA is that BRA instruction jumps relative (the jump over an OFFSET to the program counter).

The JMP instruction does not work the same way: Let us show it with an example:

```
JMP     $FC00D2
```

It always jumps to a fixed address. So the BRA instruction is used most often when it comes to jumping within the same program, while the JMP is most used to jump outside the program (to another program).

Next instruction in the list is JSR. JSR means JUMP TO SUB ROUTINE. Until now we have only used BSR instruction because we have only jumped to sub-routines, which lies within our own program. If we perform a jump to a sub-routine, which is beyond our program, we uses the JSR. Besides being able to jump to a permanent address as the JMP instruction can, the JSR can be used indirectly with an address register. There may also be an offset specified in addition to this. Here are some examples:

```
JSR     $5000
```

and an example of an indirect jump ...

```
JSR     (A0)
JSR     20(A0)
```

The first example performs a jump to address $5000, the second example uses the value that is stored where the address in A0 points to, as the jumping address. So: If A0 contains $10000 and at that address the value $50000 is stored the jump will to address $ 50000.

In the last example the jump will got to the address located where A0 + 20 points to. In other words, it takes the value located at address $10000 + 20 = $10014, and uses this value as the jump destination address.

The next instruction is BCHG and means BIT CHANGE. This instruction is used to invert a special BIT at an address or in a register. Here are some examples:

```
D0 = %00101101
```

we do ...

```
BCHG #3, D0
```

... and D0 becomes:

```
D0 =%00100101
```

We perform the same instruction a second time ...

```
BCHG #3, D0
```

... and D0 are again:

```
D0 =%00101101
```

This instruction should be obvious. The next instruction, we already have explained a part in the program example MC0901, namely RTE. RTE does ReTurn from Exceptions. We should not overwhelm you too much trying to explain what EXCEPTIONs are in this chapter, but be will explain how the instruction works. RTE works like the RTS instruction which is used at the end of a routine. The difference is that we use the RTE to complete an interrupt routine. The technical difference is that the RTS instruction just downloads the old program counter from STACK while the RTE instruction additionally retrieves the old value of the STATUS register.

The last two instructions are ROR and ROL which means “ROtate Right” and “ROtate Left” respectively. These instructions are very similar to the LSR and LSL. The only difference is that the BIT which falls out at one end comes back in at the other end of the BIT group.

Let us again show some examples:

```
D0 =%11010010
```

We try this ...

```
ROR.B     #3, D0
```

... and D0 now becomes this:

```
D0 =%01011010
```

We take one to:

D0 =%0110100101011011

after the following is done ...

ROL.W #1, D0

... D0 is this:

D0 =%1101001010110110

It may also be mentioned that 8 is the number value of rotations. If you need to rotate the for example 13 BITS, you can you either use two instructions, or take advantage of the following

Method:

```
MOVEQ     #13, D1
ROL.L     D1, D0
```

We hope it was understandable - otherwise try one more time.

## SOLUTIONS FOR TASKS IN ISSUE VIII

Task 0801: Amplitude is the height (strength) of a sound wave.

Task 0802: Sampling is the conversion of analog audio waves into a digital representation which can be used by computers.

Task 0803: An A/D converter is an electronic device, which converts an analog signal to a (binary) data.

Task 0804: Result in D1 is 225 ($000000E1)

## TASKS FOR ISSUE IX

Task 0901: What is the advantage of making use of interrupts?

Task 0902: How are the interrupts of level 7 called in the AMIGA?

Task 0903: What is the BCHG instruction for?

Task 0904: What will D1 contain after those instructions are carried out (try without K-Seka):

```
MOVE.L      #$64E918AB, D1
ROR.L #3, D1
ROL.W #1, D1
ROR.B #5, D1
```

## LITTLE DATA DICTIONARY

| | |
|---|---|
| AMPLITUDE | Amplitude is the height of crest of a sound wave. |
| SAMPLE | A sample are digitally stored data as a representation of an analog sound wave. |
| A/D CONVERTER | An A/D converter is an electronic device used to convert an analog signal to a digital (binary) signal. |
| DIGITAL | Digital is a term for a signal which can be either 0 or 1 (on or off). |
| ANALOG | An analog signal is a signal that can have infinitely many values. |
| MIDI | MIDI is an acronym for Musical Instrument Digital Interface. Such a system is used for example to have one instrument manage another. MIDI is most widespread in keyboard (electric piano, synthesizer, organs, etc.) |
| INTERRUPTS | The word interrupt is self-explanatory. An interrupt in context with computers can be used to temporarily stop the execution of one program and work on another, e.g., for handling I / O devices. |
| I/O | I/O means INPUT/OUTPUT meaning all devices connected with your computer that either receives or transmit data IN or OUT of the computer's memory (eg. a disk) |
| NONMASKABLE | This word is used in context with INTERRUPTS. A non maskable interrupt is an interrupts, which can not be disabled. |
| PRIORITY | PRIORITY word can be translated as "privileged position". This is often used in context with interrupts. An interrupt with higher priority is considered to be more important than one with a lower priority. |
| VECTOR | A vector is actually a "pointer" and therefore holds an address to a subroutine or a data table. |

## COMMENTS ON ISSUE IX

Now when you are finished with this issue you have been through the program in AMIGA machine code. We assume that since you have come so far that you are now able to perform a lot of sophisticated things on your favorite machine in pure machine code.

In this issue we have reviewed interrupts. As you going to use this feature in your programs, you will learn that it is a very sensitive function. It is therefore easy that things will go wrong if you are not careful (GURU MEDITATION!). It is therefore important to keep the AMIGA correctly working. Make sure that you always have a backup of your sources before you try to start your code.

Do not forget to debug (remove errors) your program while you are programming. It is almost impossible to spot all errors – and where they occurred – especially when your program has become so large that it covers several A4 pages.

Also make sure you comment your source code as good as possible so that you always know what the different routines are for. Using comments makes the code readable and easy to understand. It may be difficult to remember the short words like "sprcol" which, e.g., means Sprite Collision (or is it SPRITE-COLOR?) when you return to the location in your code after you have worked on other parts.

It is also important that you keep a regular programming pace – in other words: strike while the iron is hot. It is easier to write manageable code while the structure is simpler to debug.

In the next issue – issue X - we come to a whole new area namely the AMIGA's operating system, memory allocations, reading / writing files from and to disk and read / write to CLI. These are the most important and most widely used system functions.

It looks like that one rarely will need more system features than those we have mentioned here. And that’s precisely the reason why it is important to be able to use them properly. In addition it will be easier to understand how other system functions work when (or if?) you are seeking out specialized literature on the subject.

We hope that you continue to enjoy!