Practical 3
Stack and Subroutines

Prerequisite: students are required to have read the lecture notes from page 27 to page 32.

Step 1

In this step, we are going to turn the **`Abs`** program from the previous practical, which returns the absolute value of a signed integer, into a subroutine.

- **Type the program** below and **save it** under the name "`Abs.asm`".

```
            ; ==============================
            ; Vector Initialization
            ; ==============================

            org     $4

vector_001  dc.l    Main

            ; ==============================
            ; Main Program
            ; ==============================

            org     $500

Main        move.l  #-4,d0
            jsr     Abs

            move.l  #-32500,d0
            jsr     Abs

            illegal

            ; ==============================
            ; Subroutines
            ; ==============================

Abs         tst.l   d0
            bpl     \quit

            neg.l   d0

\quit       rts
```

This source code is divided into three parts:

- Vector initialization.
- Main program.
- Subroutines.

Remember this structure. When you design your own subroutines, they will have to be tested this way; that is to say, they should be called from a main program. Therefore, you always have to write a main program that calls your subroutine once or several times with different test values.

Your main program has to end with a breakpoint and your subroutines with an `RTS` instruction.

In this example, the main program contains two calls to the **`Abs`** subroutine with two different test values and ends with an `ILLEGAL` instruction breakpoint.

- It is noteworthy that a backslash character (\) can be found just before the `quit` label. What does this mean?

  First of all, a label that is not preceded by a backslash character is absolute. For instance, the **`Main`** and **`Abs`** labels are absolute. Moreover, only one single name is possible for an absolute label in a source code. If a second **`Abs`** label is found during the assembly process, the assembler will return an error.

  A label that is preceded by a backslash character is relative to the first absolute label above. For instance, the `\quit` label is relative to the **`Abs`** label. In fact, the assembler considers that the full name of the `\quit` label is `Abs\quit`.

  The advantage of relative labels is that programmers are allowed to use multiple labels that have the same name. For instance, each subroutine can have its own relative label `\quit`.

  Multiple relative labels (`\loop`, `\quit`, etc.) can be associated with the same absolute label.

  Since our source file contains only one `\quit` label, we did not have to make it relative. However, from now on, we will comply with the two following rules:

- **The label of a subroutine will always be absolute.**
- **The labels inside a subroutine will always be relative.**

- **Assemble the program** and **launch the debugger**.

- **Press `[F11]`**. **D0** has been initialized.

- The next instruction is a call to the **`Abs`** subroutine (`JSR $51A`). The return address is the address of the instruction that follows the `JSR`. That is the address `$50C`.

- **Press `[F11]`**. We are going into the subroutine.

  The return address has been pushed onto the stack.

  The stack is symbolized by the list on the left-hand side of the window. The top of the stack is just below the grey part of the list, which shows the data that is no longer on the stack. The stack can be displayed with 8-, 16- or 32-bit words.

- **Execute each instruction up to the `RTS` (excluded).**

  The **D0** register is now `4`, which is the absolute value of `-4`.

- **Execute the `RTS`.**

  The **PC** register points to the instruction that follows the `JSR`. The return address has been popped off and can be seen in the grey part of the list.

- To execute the last instructions, we are going to use the **`[F10]`** key instead of the **`[F11]`** key.

- **Press `[F10]`.**

  D0 has been initialized (same result as a press of **`[F11]`**).

- **Press `[F10]` again.**

  The whole subroutine has been executed. The return address is no longer on the stack and **D0** holds the result.

  The **`[F10]`** key allows executing a whole subroutine.

`[F10]` : Step over – step to the next line displayed on the screen (runs a whole subroutine).
`[F11]` : Step into – step to the next instruction (does not run a whole subroutine).

- **Close the debugger.**

Step 2

In this step, we are going to turn the **`StrLen`** program from the previous practical, which returns the length of a string, into a subroutine.

- **Type the program** below and **save it** under the name "`StrLen.asm`".

```
            ; ==============================
            ; Vector Initialization
            ; ==============================

            org     $4

vector_001  dc.l    Main

            ; ==============================
            ; Main Program
            ; ==============================

            org     $500

Main        movea.l #String1,a0
            jsr     StrLen

            movea.l #String2,a0
            jsr     StrLen

            illegal

            ; ==============================
            ; Subroutines
            ; ==============================

StrLen      move.l   a0,-(a7)       ; Save A0 on the stack.

            clr.l   d0

\loop       tst.b   (a0)+
            beq     \quit

            addq.l  #1,d0
            bra     \loop

\quit       movea.l  (a7)+,a0       ; Restore A0 from the stack.
            rts

            ; ==============================
            ; Data
            ; ==============================

String1     dc.b    "This string is made up of 40 characters.",0
String2     dc.b    "This one is made up of 26.",0
```

- In order to improve readability, a fourth part that contains the data required by the program has been added.

- From now on, we will comply with the following rule:

Except for the output registers, none of the data or address registers must be modified when the subroutine returns.

Be careful! This does not mean that you cannot modify any data or address registers in a subroutine. It means that when a subroutine returns, the registers that are not output registers have to contain the same values as they contained when the subroutine was called.

For instance, the output register of the **`StrLen`** subroutine is **D0**. It will contain the length of the string. Therefore, it is natural that its contents are modified by the subroutine.

On the other hand, **`StrLen`** uses and modifies **`A0`**, which is not an output register. Consequently, the value of **A0** has to be restored before the subroutine returns. To do so, **A0** is pushed onto the stack at the beginning of the subroutine and popped off just before the subroutine returns. That is the reason why the `MOVE` and `MOVEA` instructions have been added.

- **Assemble the program** and **launch the debugger.**

- **Press `[F11]`**.

  A0 has been initialized to the address of the first string (**A0** = `$0000052E`).

- **Press `[F11]`**.

  We jumped to the **`StrLen`** subroutine. The return address is at the top of the stack.

- Now, we are going to push **A0** onto the stack. **Press `[F11]`**.

  We can see that the value of **A0** (`$0000052E`) is at the top of the stack.

- **Click on the border to the left of the address `$52A`**. You have just placed an address breakpoint on the `MOVEA` instruction.

- **Press `[F9]`** in order to execute the rest of the subroutine.

  D0 holds the length of the string (**D0** = 40 = $28) and **A0** has been modified.

- **Click on the border to the left of the address `$52A`** in order to remove the breakpoint.
- **Press `[F11]`**. The initial value of **A0** has been popped off the stack and loaded into **A0**.
- **Press `[F11]`** in order to return the subroutine.
- **Press `[F10]`**. **A0** has been initialized to the address of the second string (**A0** = `$00000553`)
- **Press `[F10]`**. The whole subroutine has been executed. **D0** holds the length of the string (**D0** = 26 = $1A) and **A0** has not changed.
- **Close the debugger.**

Step 3

Now, let us turn the **SpaceCount** program from the previous practical into a subroutine. This time, the suggested key modifies three registers: **D0**, **D1** and **A0**. The output register is **D0**; it is then modified. On the other hand, **D1** and **A0** have to be restored. Therefore, we are going to save them on the stack. Since several registers have to be pushed off, we can use the `MOVEM` instruction.

- **Type the program** below and **save it** under the name "`SpaceCount.asm`".
- **Execute it step by step** by using the **`[F10]`** key (step over).
- Check that the return values contained in **D0** are right.

```
            ; ==============================
            ; Vector Initialization
            ; ==============================

            org     $4

vector_001  dc.l    Main

            ; ==============================
            ; Main Program
            ; ==============================

            org     $500

Main        movea.l #String1,a0
            jsr     SpaceCount

            movea.l #String2,a0
            jsr     SpaceCount

            illegal

            ; ==============================
            ; Subroutines
            ; ==============================

SpaceCount  movem.l   d1/a0,-(a7)       ; Save registers on the stack.

            clr.l   d0

\loop       move.b  (a0)+,d1
            beq     \quit

            cmp.b   #' ',d1
            bne     \loop

            addq.l  #1,d0
            bra     \loop

\quit       movem.l   (a7)+,d1/a0       ; Restore registers from the stack.
            rts

            ; ==============================
            ; Data
            ; ==============================

String1     dc.b    "This string contains 4 spaces.",0
String2     dc.b    "This one only 3.",0
```

Step 4

Turn the **`LowerCount`** program from the previous practical into a subroutine. If you use other registers than **D0**, save them on the stack in order to restore them before the subroutine returns.

Use the following structure in order to run and test your subroutine:

```
            ; ==============================
            ; Vector Initialization
            ; ==============================

            org     $4

vector_001  dc.l    Main

            ; ==============================
            ; Main Program
            ; ==============================

            org     $500

Main        movea.l #String1,a0
            jsr     LowerCount

            movea.l #String2,a0
            jsr     LowerCount

            illegal

            ; ==============================
            ; Subroutines
            ; ==============================

LowerCount  ; ...
            ; ...
            ; ...

            ; ==============================
            ; Data
            ; ==============================

String1     dc.b    "This string contains 29 small letters.",0
String2     dc.b    "This one only 10.",0
```

Step 5

Write the **AlphaCount** subroutine that returns the number of alphanumeric characters in a string.

Input : **A0.L** points to a string whose number of alphanumeric characters is to be found.

Output : **D0.L** returns the number of alphanumeric characters in the given string.

Tips:

- An alphanumeric character is a letter (small or capital) or a digit (from 0 to 9).
- First of all, write the two subroutines below. Keep the same structure as **LowerCount** for each of them.
  - **UpperCount** that returns the number of capital letters in a string.
  - **DigitCount** that returns the number of digits in a string.
- Your **AlphaCount** subroutine should call the **LowerCount**, **UpperCount** and **DigitCount** subroutines and return the sum of their returned values.

Use the following structure in order to run and test your subroutine:

```
            ; ==============================
            ; Vector Initialization
            ; ==============================

            org     $4

vector_001  dc.l    Main

            ; ==============================
            ; Main Program
            ; ==============================

            org     $500

Main        movea.l #String1,a0
            jsr     AlphaCount

            movea.l #String2,a0
            jsr     AlphaCount

            illegal

            ; ==============================
            ; Subroutines
            ; ==============================

LowerCount  ; ...

UpperCount  ; ...

DigitCount  ; ...

AlphaCount  ; ...

            ; ==============================
            ; Data
            ; ==============================

String1     dc.b    "This string contains 42 alphanumeric characters.",0
String2     dc.b    "This one only 13.",0
```

Step 6

Write the **`Atoui`** subroutine (*ASCII to Unisgned Int*) that returns the integer value of a string of characters. We assume that the string is not null and contains only digits that make up a 16-bit unsigned integer (form 0 to 65,535). For instance, the string "52146" should return the integer value 52146.

Input : **A0.L** points to the string to convert.

Output : **D0.L** returns the integer value of the given string.

Tips:

- The integer value of a character can be obtained by subtracting the ASCII code of the '0' character.

  Example:

```
            move.b  #'6',d1     ; D1.B = $36 (ASCII code of the '6' character).
            subi.b  #'0',d1     ; D1.B = 6   ($36 - $30 = 6).
```

- A loop should get each character of the string, convert them into integers and add them in a variable that holds the result.

  Example:
  For the string "52146", a variable should take successively the following values: 5, 52, 521, 5214, 52146. A multiplication by 10 should then appear somewhere in the loop.

- Use **D1** to read a character in the string and to convert it into its integer value.

- Use **D0** to store the successive values of the result.

- Use the previous structure to run and test your subroutine (vector initialization, main program, subroutine and data). Use at least three test strings (e.g. "52146", "309" and "2570").