Practical 6
Space Invaders (Part 1)

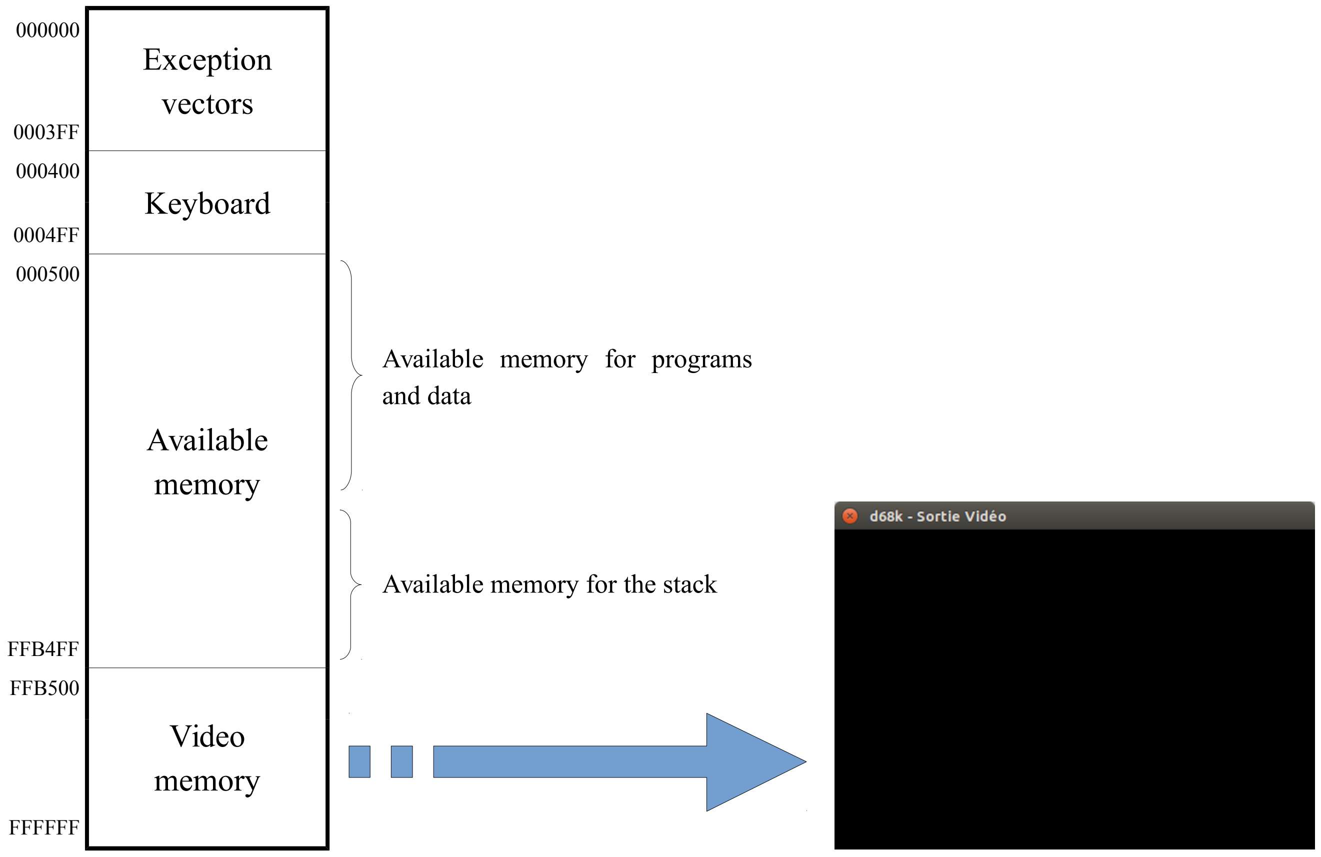
Approximate duration: 6 hrs.

In this set of practicals, you are going to develop a version of the *Space Invaders* game. You should follow the given process in order to progress step by step.

Let us start with some simple rules that you should observe as you go along the different steps:

- Your source file should be divided into five distinct parts: definitions of constants, vector initialization, main program, subroutines and data.
- Your source file should contain a single main program and several subroutines.
- The main program should make it possible to test a subroutine in progress.
- **Except for the output registers, none of the data and address registers should be modified when the subroutine returns.**

To complete these practicals successfully, you have to understand how graphics are displayed in the video output window of d68k. This type of display uses a part of the 68000 memory as video memory (a little as a graphic card would do). This video memory is located from the address FFB500$_{16}$ to the address FFFFFF$_{16}$. Here is how the memory space of d68k is organized:



About the exception vectors, only two of them will be initialized:

- **The vector 0 (located at the address 0), used to initialize the supervisor stack pointer (SSP).**

  In order to push data onto the stack while saving the largest memory space for programs, it is advisable to initialize the stack pointer to the address $FFB500_{16}$ (i.e. the starting address of the video memory). Actually, when data is pushed, the stack pointer is decremented first; therefore, the top of the stack will always be lower than the video memory. The former will never overwrite the latter. A program should then start at the address 500_{16} in order to be as far as possible from the stack pointer. This difference will be large enough to avoid any overlaps.

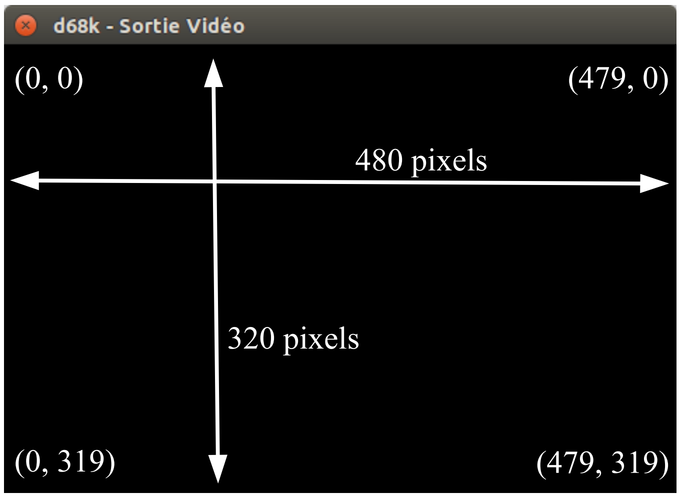
- **The vector 1 (located at the address 4), used to initialize the program counter (PC).**

  You have already used this vector in previous practicals. As a reminder, it holds the entry point of the program, which will be loaded into the **PC** register after the 68000 has been reset.

The memory space from 400_{16} to $4FF_{16}$ is reserved for keyboard handling and will be explained when we need to detect any pressed keys. For the time being, you just have to know that this memory space cannot be written.

Now, let us focus on the video memory. The main principle is quite simple: one bit of the video memory is paired with one pixel of the video output window. If a video bit is 0, its associated pixel is set to black. If a video bit is 1, its associated pixel is set to white.

A pixel has coordinates (abscissa, ordinate). The resolution in pixels of the output window is 480 × 320 (width × height):



The first video address ($FFB500_{16}$) holds eight bits, so eight pixels. These are the eight pixels at the top left corner. The next address ($FFB501_{16}$) holds the next eight pixels and so on and so forth up to the end of the first line (ordinate = 0).

Address :	**$FFB500_{16}$**								**$FFB501_{16}$**							
Pixel :	0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0	10,0	11,0	12,0	13,0	14,0	15,0

A line is made up of 60 bytes (480 / 8). To determine the address of the second line (ordinate = 1), 60 ($60_{10} = 3C_{16}$) must be added to the address of the first line, the same goes for the second line to the third line, etc. The table below gives an overview of the video addresses:

Line	**Address (in hexadecimal)**				
0	FFB500	FFB501	FFB502	…	FFB53B
1	FFB53C	FFB53D	FFB53E	…	FFB577
2	FFB578	FFB579	FFB57A	…	FFB5B3
…	…	…	…	…	…
318	FFFF88	FFFF89	FFFF8A	…	FFFFC3
319	FFFFC4	FFFFC5	FFFFC6	…	FFFFFF

For example:

- The pixel (0, 0) is the bit 7 of the address $FFB500_{16}$.
- The pixel (0, 319) is the bit 7 of the address $FFFFC4_{16}$.
- The pixel (479, 0) is the bit 0 of the address $FFB53B_{16}$.
- The pixel (479, 319) is the bit 0 of the address $FFFFFF_{16}$.
- The pixel (3, 2) is the bit 4 of the address $FFB578_{16}$.
- The pixel (21, 318) is the bit 2 of the address $FFFF8A_{16}$.

In order to avoid manipulating numerical values, we are going to define constants by using the `EQU` directive (see Chapter 1). For instance, the `VIDEO_START` constant will be assigned to the starting address of the video memory so that the address $FFB500_{16}$ appears only once in the source code. That way, whenever you need it, use the constant instead of the starting address of the video memory; it is easier.

Use the following structure for your source code:

```
                    ; ==============================
                    ; Definitions of Constants
                    ; ==============================

                    ; Video Memory
                    ; ------------------------------

VIDEO_START         equ     $ffb500                         ; Starting address
VIDEO_WIDTH         equ     480                             ; Width in pixels
VIDEO_HEIGHT        equ     320                             ; Height in pixels
VIDEO_SIZE          equ     (VIDEO_WIDTH*VIDEO_HEIGHT/8)    ; Size in bytes
BYTE_PER_LINE       equ     (VIDEO_WIDTH/8)                 ; Number of bytes per line

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

                    org     $0

vector_000          dc.l    VIDEO_START                     ; Initial value of A7
vector_001          dc.l    Main                            ; Initial value of the PC

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

                    org     $500

Main                ; ...
                    ; ...
                    ; ...

                    illegal

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

                    ; ...
                    ; ...
                    ; ...

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

                    ; ...
                    ; ...
                    ; ...
```

Step 1

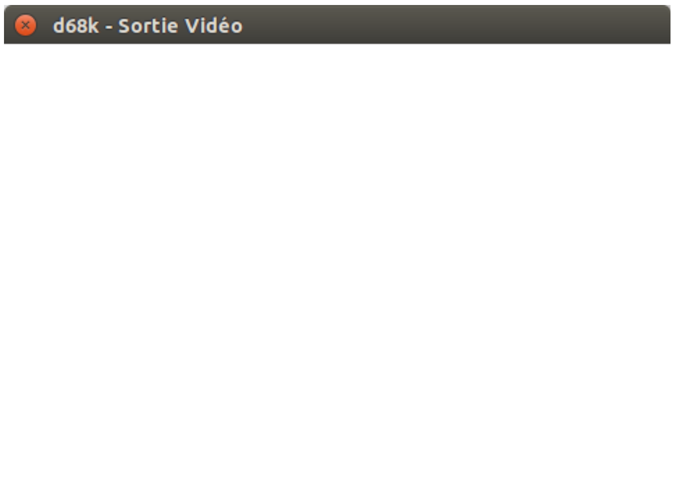
In this step, you are going to start by something very simple in order to assimilate the structure of the video memory. Write the **FillScreen** subroutine that fills the video memory with a 32-bit integer.
Input : **D0.L** = A 32-bit integer used to fill the video memory.

Use the main program below in order to run and test your subroutine. To call the subroutine, press **[F10]** or keep **[F11]** pressed down for a slower execution allowing you to see the changes on the screen and have a better understanding of your program. Be careful, if you press **[F9]**, the emulator will be too fast and you will not have enough time to see how the screen is changing. As a reminder, press **[F4]** to show the video output window.

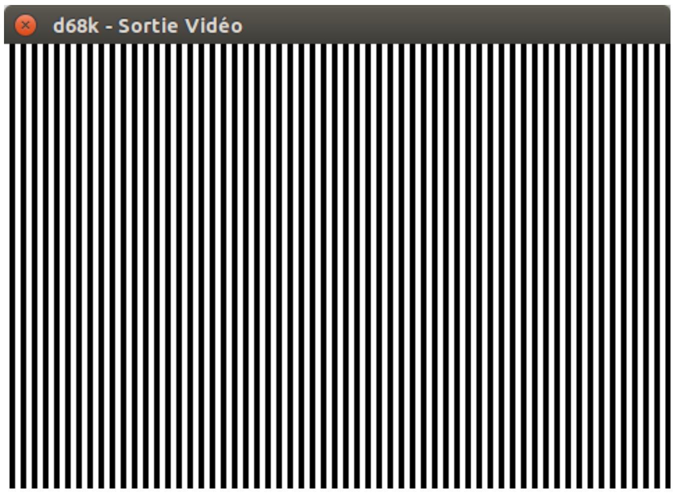
```
Main                ; Test 1
                    move.l  #$ffffffff,d0
                    jsr     FillScreen

                    ; Test 2
                    move.l  #$f0f0f0f0,d0
                    jsr     FillScreen

                    ; Test 3
                    move.l  #$fff0fff0,d0
                    jsr     FillScreen

                    ; Test 4
                    moveq.l #$0,d0
                    jsr     FillScreen

                    illegal
```

Check that your screen is identical to the following screenshots for each test:

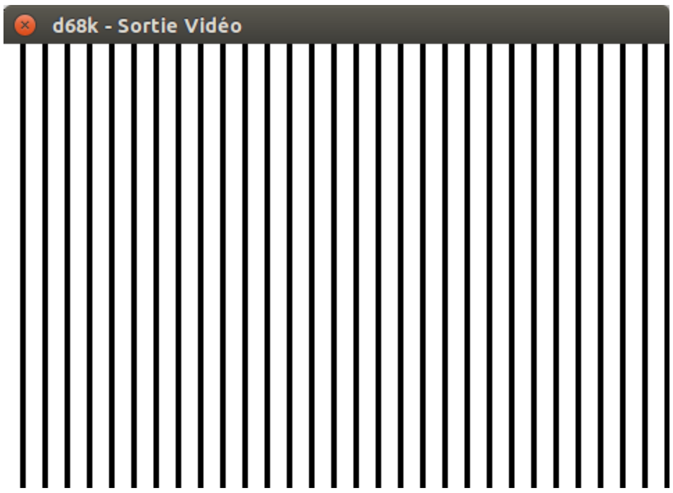
Test 1
(white screen)



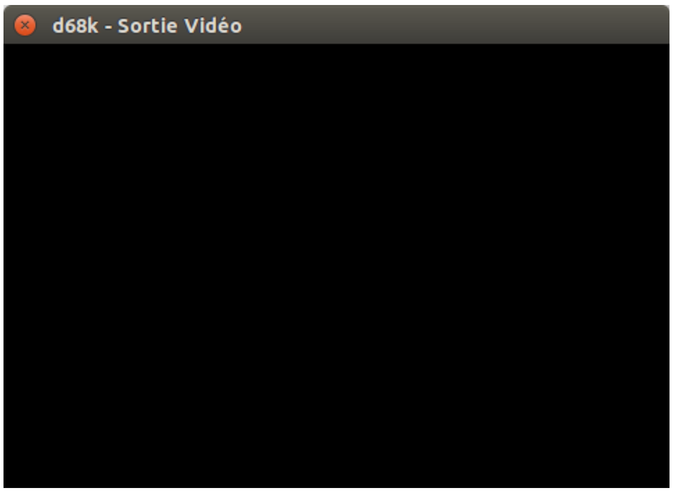
Test 2
(black and white stripes)



Test 3
(narrow black and large white stripes)



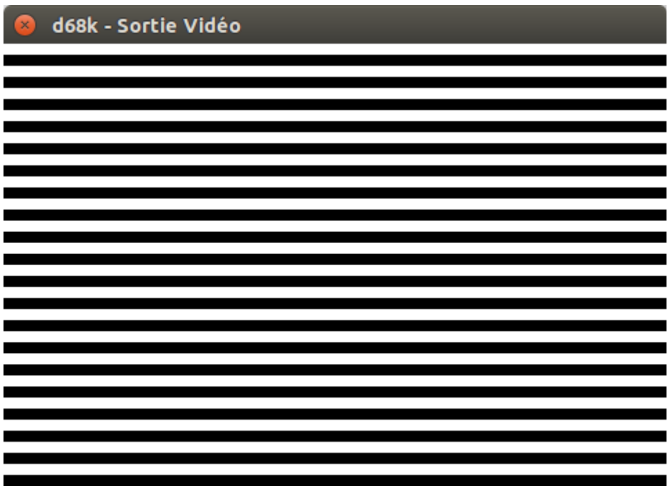
Test 4
(black screen)



Step 2

Write the **`HLines`** subroutine that draws horizontal black and white stripes. The height of each stripe should be 8 pixels.

Screenshot of the expected result:

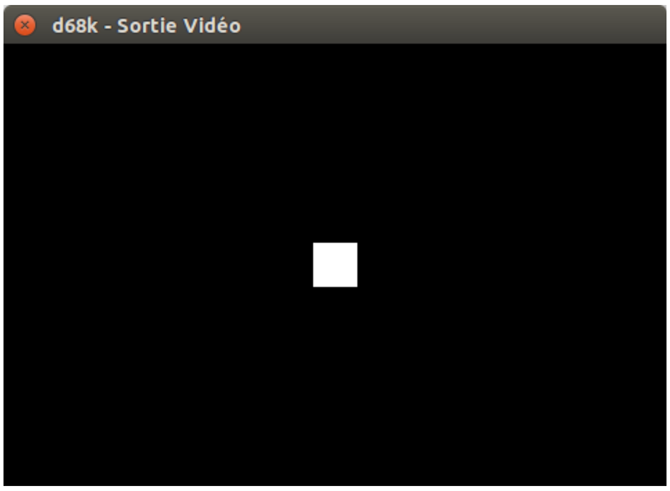


Step 3

1. Write the **`WhiteSquare32`** subroutine that draws a white square in the middle of the screen. The size of the square is 32 by 32 pixels.

   Instructions:
   The starting address of the square should not be computed dynamically (i.e. computed by the program running). You should compute it yourself and write its value directly in the source code.

Screenshot of the expected result:



2. Write the **`WhiteSquare128`** subroutine that draws a white square in the middle of the screen. The size of the square is 128 by 128 pixels. Follow the same instructions given previously.

3. Write the **`WhiteSquare`** subroutine that draws a white square in the middle of the screen. The size of the square will be passed as a parameter when calling the subroutine.
   Input: **D0.W** = Size of the square in bytes (the size in pixels is then a multiple of 8).

   - **Note:**
     Since the height and width of the window are even, the square cannot be centred if **D0.W** is odd. For the time being, ignore this.

   - **Instructions:**
     In order to avoid nested loops and to limit the size of **`WhiteSquare`**, it is advisable to write the **`WhiteLine`** subroutine that draws a single horizontal white line. **`WhiteLine`** can then be called by **`WhiteSquare`** in a simple loop. The **`WhiteLine`** inputs are as follows:
     Inputs: **A0.L** = Starting video address of the line.
     D0.W = Size of the line in bytes.

   Use the main program below in order to run and test **`WhiteSquare`**. Execute this program by pressing **`[F10]`** successively.

```
Main                move.w  #2,d0

\loop               jsr     WhiteSquare

                    addq.w  #2,d0
                    cmp.w   #40,d0
                    bls     \loop

                    illegal
```