VGA Invaders

Allan Sousa
MEIC
FEUP
Porto, Portugal
up201800149@up.pt

Breno Accioly
MEIC
FEUP
Porto, Portugal
up201800170@up.pt

Diana Freitas
MEIC
FEUP
Porto, Portugal
up201806230@up.pt

Juliane Marubayashi
MEIC
FEUP
Porto, Portugal
up201800175@up.pt

***Abstract*—The present report describes the work developed in the Embedded and Real Time Systems curricular unit. In this project, a VGA signal was successfully generated with the support of an Arduino Due to display and compute a game based on the classic Space Invaders while following strict timing constraints.**

***Index Terms*—VGA, Real Time Systems, Arduino, Cyclic executive scheduling**

I. Introduction

Video Graphics Array (VGA) is a video display controller introduced in the late 80s, which allows displaying an image in a monitor. Although, the signal is usually generated and emitted by chips, it is also possible to use a micro-controller. In this project, in particular, an Arduino Due was used.

Moreover, a simple game based on the Space Invaders was implemented, with its logic divided in distinct tasks which were coordinated using a cyclic executive scheduling approach.

This paper starts by presenting the background, in section 2. Then, section 3 details the requirements to generate the signal and section 4 is devoted to the system's architecture, describing the tasks, their scheduling and the hardware used.

II. Background

A common resolution used to emit VGA signals is 640x480 with a refresh rate of 60Hz. In other words, the produced image is represented as a matrix of 480 lines and 640 columns, with a refresh rate of 60 frames per second.

To generate and display an image using VGA, restrict time constraints need to be followed, and five different signals need to be outputted by the Arduino: the VSync, HSync, and three other signals for each color of RGB. The HSync is a pulse generated to signalize the beginning of each line. On the other hand, the VSync pulse tells the monitor to reset to the beginning of the screen ate the end of each frame.

Table I shows the general timing constraints associated with VGA, and the tables II and III show the constraints related to the vertical and horizontal synchronization signals.

TABLE I: General Timing

Screen refresh rate	60Hz
Vertical refresh	31.470 KHz
Pixel freq.	21.175 Mhz

TABLE II: Horizontal Timing

	Timing [µs]
Visible Area	25.42
Front Porch	0.64
Sync Pulse	3.81
Back Porch	1.91
Whole line	31.75

TABLE III: Vertical Timing

	Timing [ms]
Visible Area	15.25
Front Porch	0.32
Sync Pulse	0.07
Back Porch	1.05
Whole line	16.68

III. Requirements

Before defining the tasks required in this project, it was necessary to guarantee that, using the selected board, it was feasible to fulfill the strict time constraints that are required to generate the VGA signal while also managing the user interaction and updating the state of the game accordingly. To do so, a study regarding the VGA timings was performed:

Given the refresh rate (60@Hz) and considering the information portrayed in figure 1, the first step was to determine the period of the Vertical and Horizontal synchronization signals. The Vertical synchronization occurs at the end of each frame, which means that the period of the VSync is the following:

$$T_{Vsync} = \frac{1}{60} \simeq 16.67\text{ms}$$

The Horizontal synchronization occurs at the start of every line, which means that the period of the HSync can be determined by dividing the period required to generate the frame by the number of lines, which sum up to 525 if we consider the sync pulse and the front and back porches.

$$T_{Hsync} = \frac{T_{Vsync}}{480 + 2 + 33 + 10} = \frac{T_{Vsync}}{525} \simeq 31.75\text{µs}$$

Knowing that 31.75µs are required to draw one line, and that each line corresponds to 800 pixels, if we include the sync pulse and the front and back porches, then we must draw a pixel every 39.68 ns

$$T_{pixel} = \frac{T_{Hsync}}{640 + 96 + 48 + 16} = \frac{T_{Hsync}}{800} \simeq 39.68\text{ns}$$

As we used an Arduino Due, which has a frequency of 84MHz, the clock period is 11.9ns (1/84Hz). Since the Arduino has a higher frequency than the pixel draw rate, we concluded that this project is feasible. However, the time to run other tasks is minimal.

Following Figure 1, we have the time measures to update:

- The **whole line**: $800 * 39.68ns \simeq 31.744\mu s$;
- The **visible part** of a line: $(48 + 640) * 39.68ns \simeq 27.299\mu s$.

Thus, the time left to run other tasks is: $31.744\mu s - 27.299\mu s = 4.445\mu s$. These tasks are responsible for updating the information on the screen, and this is a short time to handle updates. For this reason, we have chosen to improve the speed of our tasks, by printing a pixel twice and consequently rendering a screen with a resolution of 320x240.

This change doesn't affect the monitor timings, only the task's execution time, since now the matrix representing the monitor is reduced to a quarter.

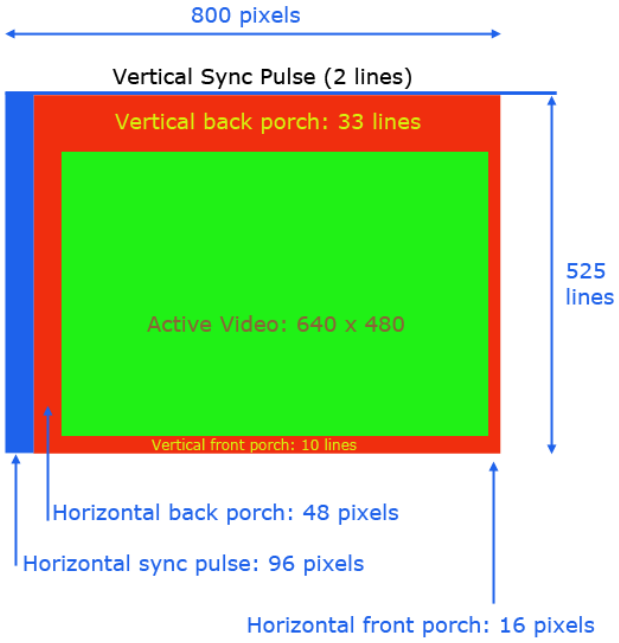

Fig. 1: VGA Timing

IV. System Architecture

The system is composed of an Interrupt Service Routine and several periodic tasks, sharing the same memory space, which is used to store and control the game behavior. The overall architecture is presented in Figure 2.

It is also worth mentioning that there isn't the possibility of race conditions since there isn't a variable modified by two tasks simultaneously. Thus, the use of synchronization mechanisms wasn't necessary.

A. Timer interrupt

To output a VGA signal with Arduino Due, an interrupt was configured to execute every 31.75µs, which corresponds to the period of the horizontal synchronization. To achieve

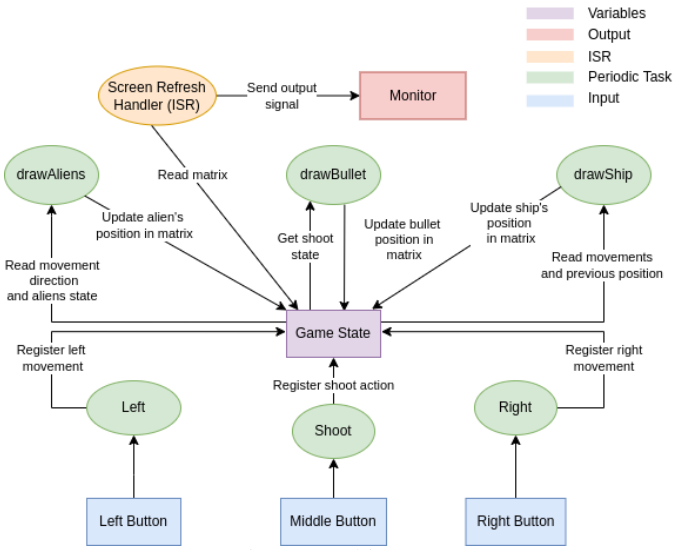

Fig. 2: Architecture

this, one of the counter timers, TC0, was used to set up an interrupt service routine with the appropriate handler. Furthermore, multiple registers had to be configured to generate the HSync signal on pin 2 through the PWM (Pulse Width Modulation) technique, a method for generating an analog signal using a digital source. Particularly, the `REG_TC0_RC0` and `REG_TC0_RA0` were used to set the frequency and the duty cycle of the horizontal synchronization:

$$RC0 = \frac{MCK}{T_{HSync}} = \frac{84MHz}{31.75} \simeq 1334\mu s$$

$$RA0 = \frac{MCK}{DutyCycle} = \frac{84MHz}{T_{HSync} - HSyncPulse} \simeq 1174\mu s$$

The timer handler is also responsible for generating the pixels of the current line. For that, it starts by using NOP instructions for the horizontal back porch, and then it uses the `REG_PIOD_ODSR` register to generate the output on the RGB pins. This register allows us to set all the RGB pins with the appropriate values in a single write operation.

The ISR is also responsible for sending the vertical synchronization signal at the end of the frame and for resetting the line counter.

B. Cyclic Scheduling

As depicted in Figure 2, beyond the Interrupt Service Routine that is responsible for generating the VGA signal, the system is also composed of 6 periodic tasks which are either responsible for drawing an element of the game or for handling the input of the user. Particularly, the following tasks were defined in the `tasks.cpp` file:

- `taskRight` Verifies if the right button is pressed and increments the column of the square (`currentCol`) as long as it does not go beyond the screen limits.
- `taskLeft` Verifies if the left button is pressed and updates the column of the square (`currentCol`) as long as it does not go beyond the screen limits.

- `taskMiddle`: Verifies if the middle button is pressed and, if no other bullet is still active, sets the initial position of the bullet, allowing the `drawBullet` to start handling its movement;
- `taskDrawShip`: Updates the position of the ship in the matrix by setting the color of the last column of the ship to the color of the screen `fb`.
- `taskDrawAliens`: Updates the position of the aliens in the matrix `fb`.
- `taskDrawBullet`: If a bullet is active, this task updates its position in the matrix. It also handles the collision of the bullet with the top of the screen or with the aliens by cleaning the bullet and the aliens that were hit from the matrix or resetting them if the user successfully hit all of them.

As the Cyclic Executive algorithm was used to schedule the periodic tasks, the first step was to determine their execution time and to define their periods, always considering that the ISR generates frequent interrupts (every 31.75µs) that will preempt the executing task. Table IV shows the execution times and periods of each task, where **C** is the worst execution time of a task with interrupts disabled and **C w/ interrupts** is the **C** with interrupts enabled. The **T** is the period.

TABLE IV: Tasks

Task	C	C w/ interrupts (µs)	T (ms)
`right`	4	28	6,12
`left`	4	28	6,12
`middle`	4	28	6,12
`drawShip`	3	27	3,06
`drawAliens`	595	2726	12,24
`drawBullet`	2	27	12,24
`TC0_handler`	24.5	-	0,031 75

The period of each task was defined by choosing the velocity of the movement and then empirically adjusting the periods to make the animation smoother. For instance, we agreed that the player should be able to cross half of the screen in only 900ms, so considering that the ship is 25 pixels wide, it should move 147 pixels/s, while moving 1 pixel in each task execution. So we set the period of the task `left` to $900/147 = 6.12$ms. We also decided that when the 4 aliens were alive, they should reach the other end of the screen in only 1.8s. As an alien needs to move 145 pixels to reach the other end of the screen, we reached a period of $1.8\text{s}/145 = 12.41$ms for the `drawAliens` task, which was adapted to be a multiple of the periods of the other tasks to facilitate the scheduling. For the remaining tasks, a similar approach was followed.

Considering these periods, a micro cycle of 3.06 ms and a macro cycle of 12.24 ms were used and, 4 cycles were defined:

- Cycle 1 executes the tasks `left`, `drawShip` and `drawAliens` in this order;
- Cycle 2 executes the tasks `right` and `drawShip` in this order;
- Cycle 3 executes the tasks `left`, `drawShip`, middle and `drawBullet` in this order;
- Cycle 4 executes the tasks `right` and `drawShip` in this order;

Most of these cycles needed a delay to fill the remaining time of the micro cycle. To define these delays we had to take into account that the ISR will not only interrupt the tasks but also the delays themselves. Having this in mind, the following calculations were performed to determine the delays:

$T_{ISR} = T_{Hsync} = 31.75$µs = period of the ISR
C_{ISR} = execution time of the ISR $\in [24.3, 24.8]$µs
$D0$ = desired delay
$i = D0/T_{ISR}$ = maximum interruptions during the delay

Considering that the ISR takes 24.4µs to execute and that the desired delay is 3000µs, then the delay could be interrupted $i = \frac{3000}{T_{ISR}} \simeq 94.5$ times, which means that, in the code, we should set a delay of:

$$D1 = D0 - i * C_{ISR} = 3000 - 94.5 * 24.4 \simeq 695\text{µs}$$

After setting up the delay, the duration of the cycles were measured and the cycles proved to endure around $3.06ms$.

C. Hardware

The system shown in Figure 3 is composed of an Arduino Due (serving as the main control unit), three buttons (serving as input to the game), and a VGA adapter with only three channels. The VGA cable expects 5V and a ground connection.

V. Conclusion

To sum up, we were able to verify that the proposed schedule works, and we were indeed able to draw a 320x240 frame through VGA with the 84MHz Arduino Due while following the timing constraints. The architecture of the system was also suitable to model the system at hand.

To follow up on the work that has been done so far, a more powerful micro-controller could be used to achieve an higher resolution, or we could experiment with sprites.

VI. Member contribution

- Allan Sousa 25%
- Breno Pimentel 25%
- Diana Freitas 25%
- Juliane Marubayashi 25%

VII. Video

The video of the project is available here.

References

[1] https://hackaday.io/project/9782-nes-zapper-video-synth-theremin/log/32271-vga-sync-generation, last accessed at 14/06/2022.
[2] https://www.gammon.com.au/forum/?id=11608, last accessed at 14/06/2022.
[3] https://forum.arduino.cc/t/manipulating-port-c-on-arduino-due/380088/7, last accessed at 16/06/2022.
[4] https://ww1.microchip.com/downloads/en/devicedoc/atmel-11057-32-bit-cortex-m3-microcontroller-sam3x-sam3a_datasheet.pdf, last accessed at 18/06/2022.
[5] http://tinyvga.com/vga-timing/640x480@60Hz, last accessed at 30/06/2022.
[6] https://forum.arduino.cc/t/arduino-due-pwm-interrupt/509284/6, last accessed at 28/06/2022.
[7] https://forum.arduino.cc/t/vga-output/127935/7, last accessed at 17/06/2022.

APPENDIX

A. Hardware

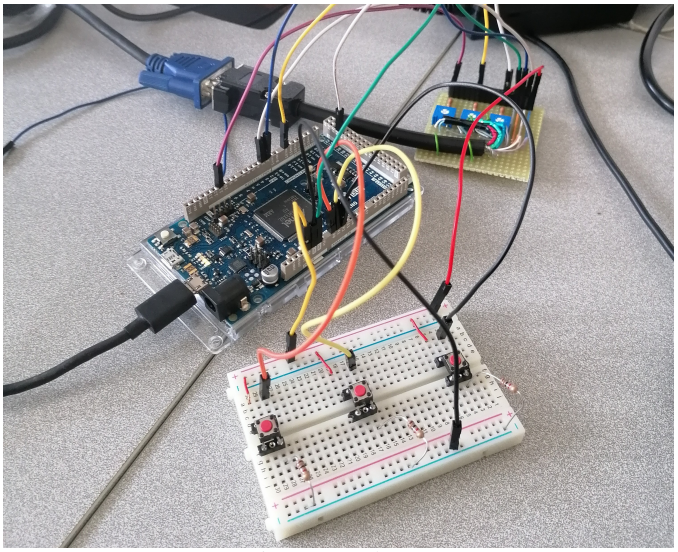

Fig. 3: Hardware