POLITECNICO DI TORINO
Department of ELECTRONICS AND TELECOMMUNICATIONS

Master Degree in Mechatronic Engineering
Electronic Systems for Sensor Acquisition - Prof. Vacca

Control of a player inside the videogame "Temple Run 2" using a multi-sensor board

Group 03

Sebastiano Ungolo - 286850
Vittorio Mayellaro - 291072
Francesco Picciocchi - 281411

January 30, 2022

Objective

This project is aimed at controlling a player inside a videogame, using the output of the sensors available on the Nucleo board shield. There are four steps to follow:

- acquiring the sensor data from the board;
- processing the data on the Nucleo using C language;
- sending the data to the computer through the serial interface;
- converting the data received from the serial interface to a Keyboard command and use it to control the player inside the videogame;

Project introduction, game and commands

The project consists in controlling a player inside the video game Temple Run 2 using a sensor shield as a joystick.
Temple Run 2 is an online video game based on an endless runner that has to dodge obstacles increasing the traveled distance and so the score. In figure 1 are shown the basic commands. In particular, pressing the up arrow on the keyboard the player can jump, with the down arrow it rolls, and with the left and right arrows it can move on the sides of the path. During the gameplay there are coins to collect to fill an energy bar which, when full, makes possible to activate the boost, pressing the space bar, making the player immortal and faster for some seconds.

Figure 1: Player movements in the game

Basically, the idea is to replace the commands from the keyboard with the movement of the sensor shield communicating with the PC through the serial port.
Furthermore, in this project, the electronic board Nucleo STM32F401RE and the sensor shield Nucleo IKS01A3 are employed.

Acquisition system

The scheme of the data acquisition system is reported in figure 2.

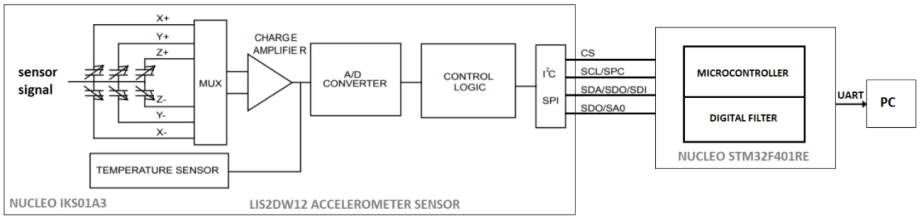

Figure 2: Scheme of the data acquisition system

The first stage of the data acquisition system is an accelerometer, which acquires the linear acceleration components along three axes. Subsequently, a multiplexer manages the data selection and a charge amplifier make the signals compatible with the ADC. The ADC converts the analog signals into discrete ones, that can be interpreted by the processor. The sensor shield's output are then sent to the Nucleo STM32F401RE through the I2C communication bus. Following the processing and a proper filtering of the data, the Nucleo board can communicate with the PC through the UART communication protocol. A python script allows the replacement of the commands from the keyboard with the output signals coming from the board. The most relevant task is the development of the code to be downloaded on the microprocessor. By means of a timer, the accelerometer signals are acquired each sampling time. Those samples are properly filtered in order to have smoother signals in the time domain. The STM32Cube MX software is used to configure the Nucleo STM32F401RE and in order to communicate properly with the sensor shield IKS01A3. Furthermore, the STM32Cube IDE software is used in order to process the data using C language whereas the STM32 Monitor is employed to evaluate the variable behaviour over time.

Sensor Shield IKS01A3

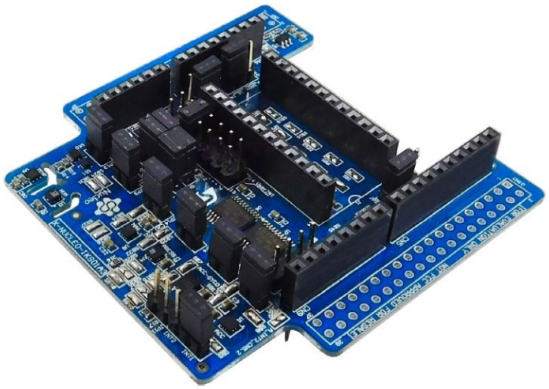

Figure 3: Photo of the Sensor Shield IKS01A3

The X-NUCLEO-IKS01A3 is a motion MEMS and environmental sensor evaluation board system. The board must be connected on the matching pins of any STM32 Nucleo board with the Arduino UNO R3 connector. The sensors present on the board are:

- LSM6DSO: MEMS 3D accelerometer ($\pm 2/\pm 4/\pm 8/\pm 16g$) + 3D gyroscope ($\pm 125/\pm 250/\pm 500/\pm 1000/\pm 2000dps$)
- LIS2MDL: MEMS 3D magnetometer ($\pm 50gauss$)
- LIS2DW12: MEMS 3D accelerometer ($\pm 2/\pm 4/\pm 8/\pm 16g$)
- LPS22HH: MEMS pressure sensor, 260-1260 hPa absolute digital output barometer
- HTS221: capacitive digital relative humidity and temperature
- STTS751: Temperature sensor (–40 °C to +125 °C)

It is important to highlight that PB9 and PB8 need to be manually set to I2C1_SDA and I2C1_SCL respectively, since the extension board uses these pins to communicate with the Nucleo on the I2C bus.
The LIS2DW12 is an ultra-low-power high-performance three-axis linear accelerometer belonging to the "femto" family and has a dedicated internal engine to process motion and acceleration detection. In figure 4 it is represented the top view of the sensor with the three axes along which the signals are acquired.

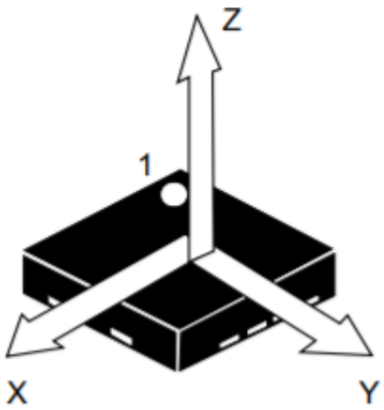

Figure 4: Direction of detectable acceleration

In figure 5 is stated the function through which it is possible to get the linear acceleration along the three axes, storing it in the struct "axes". For the sake of simplicity, these three values are stored in three different variables.

```
IKS01A3_MOTION_SENSOR_GetAxes(1, MOTION_ACCELERO, &axes);
xAxisReading = axes.x;
yAxisReading = axes.y;
zAxisReading = axes.z;
```

Figure 5: Linear acceleration acquisition along the three axes

Digital filter

Noise is a primary concern in sensors. It is possible to mitigate its effects on the output signal using filters, which are basic components of all signal processing and telecommunication systems. Digital Filters are slow and with a small dynamic range but they guarantee vastly superior performance compared to analog ones. There are two ways to get a digital filter:

- by convolution of the impulse response of the filter with the input signal (Finite Impulse Response)
- by recursion, considering also the summation of the previous output values (Infinite Impulse Response)

FIR filters are the slowest but have the best performance whereas IIR filters have worse performance but higher speed (10x).

Moving Average Filter

In this project a moving average filter, which is a FIR, is employed. It is an optimal filter for reducing white noise and smoothing signals in the time domain. It is used in this project mainly because it has good performances in the time domain and it is easy to implement. An easy implementation of the filter that causes a slight shift of the signal is the following:

$$y_{[i]} = \frac{1}{M} \sum_{j=0}^{M-1} x_{[i+j]}$$

However, in the project a slightly more complex version is employed in order to avoid further shifts:

$$y_{[i]} = \frac{1}{M} \sum_{j=-\frac{M-1}{2}}^{\frac{M-1}{2}} [x_{[i+j]}] \implies y_{[i]} = y_{[i-1]} + \frac{x_{\left[i+\frac{M-1}{2}\right]}}{M} - \frac{x_{\left[i-\frac{M-1}{2}-1\right]}}{M}$$

The moving average filter provide a noise reduction which is the square root of the number of points. In this project the number of points is 25, defined through the macro SHIFT.

FilterMovingAvg function

FilterMovingAvg is a function implemented in the project that digitally filters noisy data in input referred to an axis, as shown in figure 6. The values in input to the function are the raw accelerations data measured through the accelerometer as well as an integer value indicating one of the three axes x, y and z the measure is referred to.

```
/**
  * @brief Filter the value in input using the Moving average approach
  * @param Noisy input value to be filtered
  * @param Axis the measure is referring to. Could be :
  * - Axis x for instance 0
  * - Axis y for instance 1
  * - Axis z for instance 2
  * @retval filtered value according to the moving average approach
  */
int32_t FilterMovingAvg(int32_t noisy_value, uint8_t axis) // i=0 at the beginning
{
    int32_t old_noisy_value;
    old_noisy_value = accelero_readings[axis][i[axis]];
    accelero_readings[axis][i[axis]] = noisy_value;
    sum_readings[axis] = sum_readings[axis] + accelero_readings[axis][i[axis]]-old_noisy_value;
    filt_value[axis] = (int32_t)sum_readings[axis]/SHIFT;
    i[axis]++;
    if (i[axis] == SHIFT)
    {
        i[axis] = 0;
    }
    return filt_value[axis];
}
```

Figure 6: FilterMovingAvg function

Data comparison before and after filtering

As previously stated, the moving average filter is employed to smooth the noisy input acceleration signals referred to axes x, y and z in the time domain. In figure 7 it is possible to observe how the acceleration along the x axis evolves over time after having been filtered (orange curve).

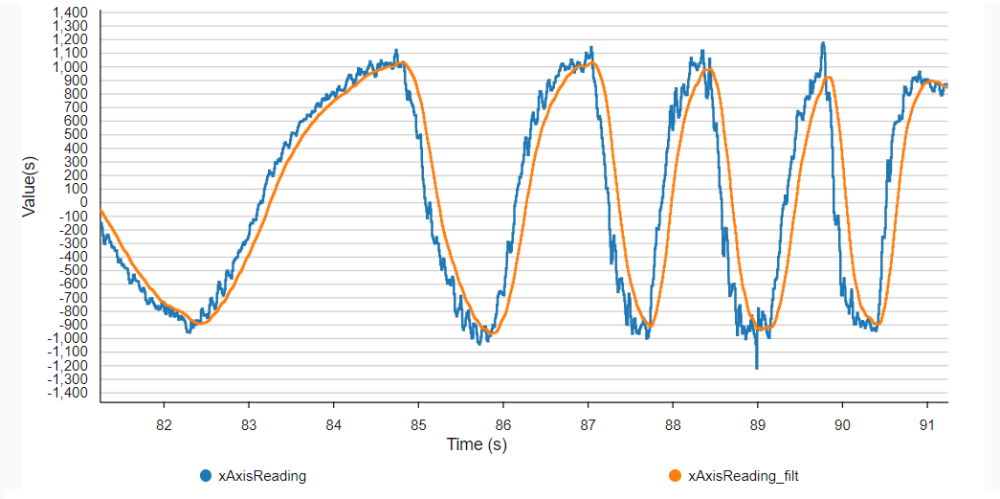

Figure 7: FilterMovingAvg function

Similar results are obtained for the acceleration along the y and z axes. It is possible to notice how this filter is suited for real time application with human interactions allowing to smooth oscillations in the signals due to noise and imperfect movements of the user.

Design choices

Sensor choice

The goal of the design is to convert physical movement of the board in the space into commands for the movement of the player in the game.
Since on the board are available gyroscope and accelerometer, which provide angular velocity and linear acceleration respectively, a possible approach could be to integer the velocity or the acceleration to get board orientation. However, this approach results to be very complex due to the drift generated by the numerical integration and for this reason it was rejected.
Therefore, a different approach is employed: in order to detect if a board movement is performed, the data coming from the sensor are compared with suitable thresholds. If the data overcomes the corresponding threshold, the related board movement has been effectively executed and a command is sent to move the player. In order to translate more movements in commands, more thresholds are used. The thresholds settings are based on a sample of eight different testers, therefore considering different gamers play styles, adapting them to the most soft one.
Referring to the gyroscope the thresholds represent angular velocities whereas, for what concerns the accelerometer, the thresholds represent linear accelerations.
This approach is more straightforward using the accelerometer instead of the gyroscope, which leads to a more complex threshold implementation in the code.

Sampling time

Considering that human movements are too slow compared with sensor rate, it is a waste of resources acquiring data from the shield continuously. Therefore, the data are acquired every 10 ms, that constitutes the sampling time T_s. Operatively, the sampling time is implemented enabling the interrupt on an internal timer of the microprocessor. The timer settings are the following:

$$PRESCALER : 840 - 1 \qquad PULSE : 1000 \qquad COUNTER\,PERIOD : 65535$$

$$f = 84MHz \implies T_s = \frac{PULSE * PRESCALER}{f} = 10ms$$

As soon as the interrupt is triggered, the pulse value is incremented by 1000 (in order to set the next interrupt) and the data acquisition is triggered in the while loop by the flag isSampling =1.

```
void HAL_TIM_OC_DelayElapsedCallback (TIM_HandleTypeDef *htim) {
// Operations to be performed each time the OC is called (every 10ms)
    if (htim == &htim3)
    {
        if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)
        {
            pulse = (HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1) + 1000);
            __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_1, pulse);
            isSampling = 1;
        }
    }
}
```

Figure 8: Output Compare Callback

Other settings

There is another operation managed by interrupts, that is the push button pressing. Indeed, pressing the blue button on the board it is possible to pause the game.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == GPIO_PIN_13 )
    {
        // Operations to be performed each time the blue button (GPIO_PIN_13) is pressed
        isBtnPressed = 1;
    }
}
```

Figure 9: Button pressing Callback

Furthermore, since there are no further tasks running during the transmit, communications on the serial port take place in polling.
In the code, all timers, push buttons, transmissions and interrupts are implemented using the HAL libraries.

Code description

The code is developed in an infinite while loop, i.e. all the instructions are cyclically repeated. At first, the acquisition of data from the sensor is scanned by the interrupt timer, opportunely set in order to have a sampling time $T_s = 10ms$. As soon as the interrupt occurs, a flag variable (isSampling) is set and the acquisition is triggered.

```
while (1)
{
    if (isSampling == 1) //TIM3 OC triggered every 10ms
    {
            isSampling = 0;
            if (k < SHIFT)
            {
                IKS01A3_MOTION_SENSOR_GetAxes(1, MOTION_ACCELERO, &axes);
                accelero_readings[0][k] = axes.x;
                accelero_readings[1][k] = axes.y;
                accelero_readings[2][k] = axes.z;
                sum_readings[0] = sum_readings[0] + accelero_readings[0][k];
                sum_readings[1] = sum_readings[1] + accelero_readings[1][k];
                sum_readings[2] = sum_readings[2] + accelero_readings[2][k];
                filt_value[0] = (int32_t)(sum_readings[0]/SHIFT);
                filt_value[1] = (int32_t)(sum_readings[1]/SHIFT);
                filt_value[2] = (int32_t)(sum_readings[2]/SHIFT);
                k++;
            }
```

Figure 10: While loop, initialization of the average value

As explained in the project introduction, the player in the game can jump, roll, shift left, shift right. The idea to replicate these commands in board movements is to take advantage of the gravitational acceleration ($\vec{g}$), since it is always present. When the board is placed on the horizontal plane, the vector $\vec{g}$ completely lays in the z axis. Nevertheless, if the board is inclined, $\vec{g}$ will have components also along the x and y axes.
Therefore, a rotation of the board around the y-axis leads to a significant variation of $\vec{g}$ contribution along the x axis. Jump and roll commands are associated with these movements. By a trial and error procedure, a positive (for jump) and negative (for roll) threshold are set on the x axis. In particular, any time the spin is sufficient to overcome the positive (negative) threshold, this will correspond to the execution of the jump (roll) command.

```
// Define the threshold used in the x, y, z Axes loop controls
#define xThreshold 300
#define yThreshold 250
#define zThreshold 1200
```

Figure 11: Definition of thresholds used to set the status of the player

Since a single runner jump (roll) is desired in correspondence of a single jump (roll) input by the gamer, an inhibition condition is put in AND logic with the threshold check condition to avoid further jumps (rolls) commands to the runner. The inhibition condition is active as soon as the first threshold overcome is detected. When the board is moved back to the rest position (below the threshold), the inhibition condition is reset.
Finally, the suitable command, JUMP or ROLL, is sent to the runner by transmitting the messages “JUMP” or ”ROLL” on the serial port where the board is connected using the serial protocol UART.

```
if(xAxisReading_filt > xThreshold && isJumping == 0)
{
    isJumping = 1; // to ensure to capture only one jump command
    sprintf(msg,"JUMP \n");
    HAL_UART_Transmit(&huart2,(uint8_t*)msg,strlen( msg),HAL_MAX_DELAY);
}
else if(xAxisReading_filt < -xThreshold && isRolling == 0)
{
    isRolling = 1; // to ensure to capture only one jump command
    sprintf(msg,"ROLL \n");
    HAL_UART_Transmit(&huart2,(uint8_t*)msg,strlen(msg),HAL_MAX_DELAY);
}
else if(xAxisReading_filt > -xThreshold && xAxisReading_filt < xThreshold)
{
    isJumping = 0;
    isRolling = 0;
}
```

Figure 12: While loop, thresholds check to send commands to the serial port

An identical approach is employed to define the left/right commands and thresholds. By the way, in this case the rotation is around the x axis and the $\vec{g}$ most significant contribution is in the y axis.
Two thresholds are defined for left and right command and also here an inhibition condition is present in AND logic to avoid sending further commands.
As for the previous case, the commands are sent to the player by transmitting on the serial port the messages "LEFT" or "RIGHT".
The last command to describe is the boost. Also in this case the idea is to use a threshold to trigger this command. A positive threshold is placed on the z axis, i.e. it is necessary to shake the board upward to counteract $\vec{g}$ and generate a positive acceleration to overcome the threshold. As usual, an inhibition condition is placed in AND logic in order to avoid sending further commands. The inhibition condition is reset during the backward motion because it generates a negative acceleration peak that for sure will overcome a specular negative threshold.
Moreover, boost command can be used even for starting a new game or to resume after a pause. In fact, the boost is activated in the game using the space bar, which is also used for other actions.

```
if(zAxisReading_filt > zThreshold && isBoosting == 0) // to ensure to capture only one boost command
{
    isBoosting = 1;
    sprintf(msg,"SPACE\n");
    HAL_UART_Transmit(&huart2,(uint8_t*)msg,strlen(msg),HAL_MAX_DELAY);
}
else if(zAxisReading_filt < zThreshold)
{
    isBoosting = 0;
}
```

Figure 13: While loop, thresholds check to activate the boost

Additional feature for the game experience is the inclusion of the pause command (ESC on the keyboard). It is implemented very simply by taking advantage of the blue push button already present on the nucleo board and it is managed by using an interrupt.

```
if (isBtnPressed==1)
{
        isBtnPressed=0;
        sprintf(msg,"ESC  \n");
        HAL_UART_Transmit(&huart2,(uint8_t*)msg,strlen(msg),HAL_MAX_DELAY);
}
```

Figure 14: While loop, button pressing check to pause the game

Communications

UART

UART, or universal asynchronous receiver-transmitter, is one of the most used device-to-device communication protocols. It uses asynchronous serial communication with configurable speed.
For UART the baud rate (maximum number of bits per second to be transferred) needs to be set the same on both the transmitting and receiving device. In this project the baud rate is set to 115200.
Here, the function HAL_StatusTypeDef HAL_UART_Transmit (UART_HandleTypeDef * huart, uint8_t * pData, uint16_t Size, uint32_t Timeout) is used to send data in blocking mode; in particular the sent data are the instruction to be translated to keyboard commands.

Conversion to Keyboard commands

Converting the data received from the serial interface to a Keyboard command and using it to control the player inside the videogame can be done using the Python module “Keyboard” by simply using the functions “keyboard.press()” and “keyboard.release()”. In figure 15 the Python code used to implement this conversion is reported.

```python
import serial
from pynput.keyboard import Key, Controller
import time
keyboard = Controller()

serial_port = serial.Serial()
serial_port.baudrate = 115200
serial_port.port = 'COM3'
serial_port.timeout = 1
error = False
time.sleep(5) #gives you time to open the desired target for the inputs
try:
    serial_port.open()
except:
    error = True

if not serial_port.is_open or error:
    print("Something went wrong!")
    exit()

while 1:
    try:
        if serial_port.in_waiting!=0:
            key = serial_port.read(6).decode('ascii')#.strip('\n').strip('\r')
            print (key)
            if key == "JUMP \n":
                key = Key.up
            if key == "ROLL \n":
                key = Key.down
            if key == "LEFT \n":
                key = Key.left
            if key == "RIGHT\n":
                key = Key.right
            if key == "ESC  \n":
                key = Key.esc
            if key == "SPACE\n":
                key = Key.space
            keyboard.press(key)
            keyboard.release(key)

    except:
        pass
```

Figure 15: Python code for commands input

It is important to highlight that the instructions from the microcontroller will be converted to keyboard commands as soon as the Python script is executed. Therefore, to deal with a proper functioning system, the Python script should be launched before starting the Temple Run 2 game.

Conclusions

Taking everything into account, the procedure adopted to develop this system can be employed as a basic framework to deal with similar games. Indeed, by introducing simple modifications in the code it is possible to extend the usability of the designed system. Plenty of test has been performed to verify the system comfortability as well as its responsiveness. In table 1 are stated the highest score of the developers. Considering that one minute of gameplay is worth about 10000 points, the scores in the table show that the system is enough comfortable to guarantee an enjoyable game experience.

Player Name	Score
Vittorio Mayellaro	39368
Francesco Picciocchi	29074
Sebastiano Ungolo	27789

Table 1: Developers highest score