CEEL - ISSN 2596-2221
Universidade Federal de Uberlândia
06 a 10 de Dezembro de 2023

JITTER-FREE AND PORTABLE HARDWARE PWM SOLUTION FOR EMBEDDED LINUX DEVICES

Vinícius Freitas Rodrigues*[1] and Éder Alves de Moura[2]

[1,2]FEELT, Universidade Federal de Uberlândia

***Abstract* - An architecture in increasing adoption for the development of larger embedded applications is the use of Linux systems. Such system appears as a basis for the development of software applications. This choice is justified by the practicality of accessing the software resources naturally made available by the system, such as the possibility of using different programming languages, network communication protocols and graphical interfaces. In addition, Linux is an open source platform and without licensing costs. However, the use of an Operating System, whose kernel is not real-time, can make the execution of some of its functions unfeasible, due to the non-preemptibility in scheduling tasks. In this sense, this work presents a study on the generation of Pulse Width Modulation (PWM) signals. The adopted approach uses an external chip, in addition to the microprocessor containing the Operating System. The proposal consists of evaluating the PWM signal being generated using an IC dedicated to PWM generation, model PCA9685, through the development of an external board, connected to the Linux device via Inter-Integrated Circuit (I^2C). The obtained results indicated that the proposal was able to generate PWM signals with stable frequency and duty cycle, however with a bias in the frequency value, later corrected via software.**

***Keywords* - PWM generation, embedded Linux, PCA9685, C++ library, Raspberry Pi.**

I. INTRODUCTION

The embedded Linux environment brings the possibility of using high-level frameworks in comparison with the standard bare-metal embedded system. It is suitable for building multi-domain applications such as with robotics, navigation algorithms, user graphical interfaces and complex control implementations.

The problem is that the most of these applications are real-time constrained, but Linux, traditionally, is not a real time operating system. The multitasking in an operating system is achieved using a scheduler, the program that decides the tasks priority and rapidly switches between programs. The most Linux schedulers are optimized for performance, unlike a Real-Time Operating System (RTOS), which prioritizes predictability [1].

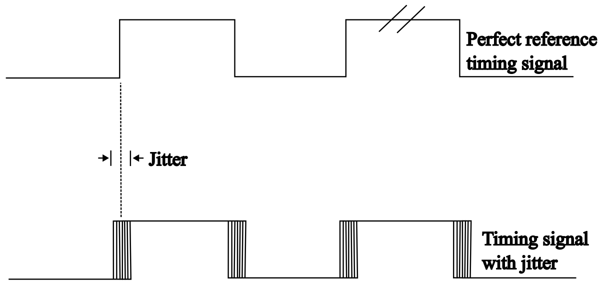
This aspect of the Linux scheduler makes its processes non-deterministic. Lack of determinism directly affects resources like the Pulse Width Modulation (PWM) signals generated by software, causing the signal's frequency and duty cycle to be unstable [2].

By definition, the pulse width modulation consists in data transmission through the variation of a square wave signal width. This information travels on the signal accordingly with the duty cycle, which is the ratio between the signal's period and the high logic pulse width [3]. As the carrier wave's period in the modulation is constant, this signal strongly depends on timing.

A reference timing digital signal, as a PWM wave, has a fixed period that does not varies over time. However, in the non-ideal world, all the signals show small variations. These variations in phase position, period and duty cycle, shown in Figure 1, are called jitter [4].

This PWM signal variation, if substantial, can affect real time applications. Many applications may be impaired by PWM jitter, for example, digital control systems, motion control and switching-mode power converters.

Figure 1: Timing signal with jitter.



On traditional operating systems, such as the Linux, the lack of preemptibility of the kernel implies that the tasks processing order are not based on timing. In this context, if a PWM signal is generated by software in a reserved task, the operating system must not process its task on a predictable timing.

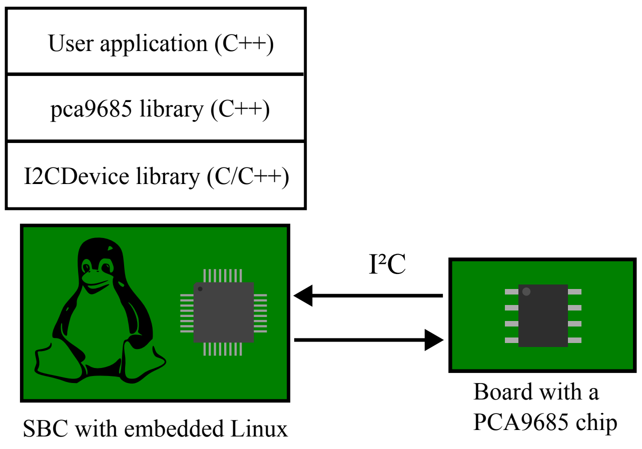
One approach for solving this problem is the implementation of the PWM using low level drivers. An example is the *ServoBlaster*, a kernel driver that implements a low level ser-

*vinicius.freitas@ufu.br

vomotor control interface for the Raspberry Pi [5]. The driver works using the Direct Memory Access (DMA) controller to generate the signal and the PWM peripheral just to set the pulse width in a way that delays are more accurate. Although elegant, this solution is hardly portable, since the DMA peripheral depends on the System-on-Chip (SoC) processor architecture.

Another approach is using an external hardware to generate the signals. In [2], for example, an external hardware module is used to create the signals. The board, named Navio2, is an open-source drone controller designed for the Rapberry Pi, and the PWM is created through a STM32 microcontroller embedded in its hardware. In this solution, a Python module was implemented to be the interface on Raspberry Pi's side. This strategy allows the integration with the some of the embedded Linux high level frameworks: ROS (Robot Operating System) and ArduPilot.

Figure 2: Scheme of the solution proposed by this work.



Adopting a strategy similar to that developed in [2], this work proposes a solution for generating a reliable PWM signal for an embedded Linux running on a Single-Board Computer (SBC), using an external hardware, the PCA9685. Furthermore, the interface with the signal generator chip was made in such a way to be portable across different hardware set-ups. Since the PCA9685 is a low price chip, it can also be done with an affordable hardware. The whole system architecture, proposed in this work, is shown in Figure 2.

II. DEVELOPMENT OF THE HARDWARE SOLUTION FOR GENERATING PWM SIGNALS

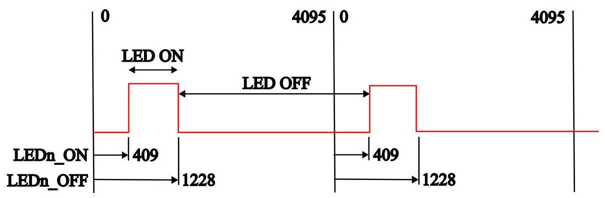
This section discuss about the hardware platform and its characteristics in the following subjects: the PCA9685 chip, the C++ interface library and the output circuit drivers.

A. *The PCA9685 chip*

In this work, the adopted external hardware for driving the PWM signals is the NXP Semicondutor's PCA9685. This chip is a dedicated IC for PWM signal generation, with 12-bit precision and an I^2C interface [6]. The IC was originally designed to be a LED controller, but its usage extends to most PWM driven circuit.

The PCA9685 have a simple register map, where a few registers menages all the chip configurations and the most of them are for controlling the 16 PWM channels. Each channel has four 8-bit registers that control their time in high or low logic level. Two of them, LEDn_ON_L and LEDn_ON_H, represent the least significant bits and the most significant bits, respectively, of the value holding the time the output must go ON. Similarly, the other two registers, LEDn_OFF_L and LEDn_OFF_H, represent the value holding the time the output must go OFF [6]. Thus, both the duty cycle and phase shift of the signal can be configured, as seen in Figure 3.

Figure 3: Example output signal on a LEDn pin.

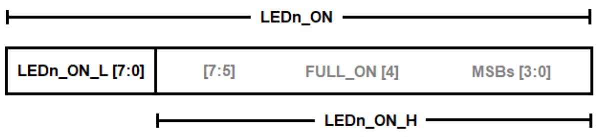


If the integrated circuit has an architecture made of 8-bit registers, then some of the bits in LEDn_ON and LEDn_OFF aren't used for value storage. For example, LEDn_ON_H and LEDn_ON_L have 8 bits each, but only 12 are used for storing the LEDn_ON time value. It happens because the registers [7:5] of LEDn_ON_H are reserved (read only) and the bit 4 is the FULL_ON bit. When the bit 4 is set, the channel output must always be in high logic level, no matter the value stored in the 12 bit value. These registers are presented in Figure 4.

The LEDn_OFF registers also follow the same operation scheme. The most significant bits of a LEDn_OFF have three reserved bits and a FULL_OFF. When the bit 4 of the LEDn_OFF_H is set, then the channel output must always be in low logic level, no matter the value stored in the 12 bit value. If the the FULL_OFF and the FULL_ON bit in the same channel are set, then LEDn_ON is ignored and the output remains in the low logic level.

The methods later implemented for enabling and disabling channels or controlling the PWM outputs are based on these LEDn registers characteristics.

Figure 4: LEDn_ON register structure. LEDn_OFF follows the same model.



Two other important registers are the MODE1 and the MODE2. Each bit in these registers configures a different PCA9586 feature. Some of these bits must always be configured before using the PWM, otherwise the signal may not be generated properly.

The minimal configuration comprises resetting the SLEEP bit (bit 4) of MODE1 and configuring the output pin topology in the OUTDRV bit (bit 2) of MODE2. If the sleep bit is set, then the internal oscillator is turned off, and so the PWM can-not operate. In addition, the output drive of the PWM pins must be set "totem-pole" (1) or "open-drain" (0) depending on the user's application.

Finally, the last essential configuration for every application is defining the PWM frequency. It can be done by setting a value in the PRE_SCALE register. The prescale value must be set according with the desired output frequency. This value can be determined with the formula shown in equation 1:

$$prescale\_value = round\left(\frac{osc\_clock}{4096 \times frequency}\right) - 1 \qquad (1)$$

The oscillator's clock frequency is $25Mhz$, if the internal PCA9685's oscillator is being used. External clock can also be supplied on the EXTCLK pin (pin 25 on TSSOP chip, or 22 on HVQFN chip). To use the external clock feature the bit 6 in the MODE1 register must be set. The prescalar can only be changed with the chip in sleep mode.

B. Interfacing the PCA9685 with a C++ library

The library [1] implements a class that wraps the PCA9685 features [7]. Since the PCA9685 interfaces with the operating system using the I^2C protocol, the IC class is inherited from the I^2C device template. Inheriting this template class, a PCA9685 object have access to the *I2CDevice* methods to perform reading or write on the bus [8].

This object-oriented approach simplifies wrapping new I^2C devices with C++ classes. It could be extended to many other devices similar to the PCA9685.

To create an object of a PCA9685 class, it is only necessary specify the I^2C bus number of the Linux device and the I^2C slave address as arguments. It is important to point out that the PCA9586's address is configured by hardware using a specific set of pins [6].

The configuring methods, like the one in Figure 5, used to set or reset MODEx bits, follow a characteristic code template. At first, the MODEx register is read and stored in a variable. Then, depending on the user's choice a bit is set or cleared in the right position. The bit position depends on the method and which configuration bit it is supposed to change. Then, the variable is written on the bus using a *I2CDevice* class method. For example, the inverting logic feature is enabled or disabled by setting or resetting the MODE2's bit in 6^{th} position.

Figure 5: Example of a configuration method.

```
void pca9685::set_output_inverting(bool invert){
unsigned char mode2;
mode2 = this->readRegister(PCA9685_MODE2);
invert? (mode2|= 1UL << MODE2_INVRT) :
        (mode2 &= ~(1UL << MODE2_INVRT));
this->writeRegister(PCA9685_MODE2, mode2);
}
```

To enable the PWM channel n, the user must call a function that resets the bit 4 in the LEDn_ON_H register. Only after calling this *enable_channel* method, the user is capable to set the desired duty cycle.

[1]Available at: https://github.com/Sr-Vinicius/pca9685_raspberry

Although the chip allows configuring both phase shift and duty cycle, the library was implemented just to control the second one. Controlling the PWM signal's duty cycle already satisfies the most of the applications. The library implements the pulse width control modifying only the values on the LEDn_ON registers. This is done by writing the eight LSBs of the unsigned short *duty_cycle* variable in the LEDn_ON_L and the four variable's MSBs in the bits [3:0] of the LEDn_ON_H. This implementation was possible by using an unsigned char pointer and pointer casting. A part of the method is presented in Figure 6.

Figure 6: Piece of code that writes the PWM duty cycle properly in PCA9685 registers.

```
unsigned char* led_on_p;
led_on_p = (unsigned char*) &duty_cycle
this->writeRegister(PCA9685_PWM_CH(channel),
                    *led_on_p);
led_on_p++;
this->writeRegister(PCA9685_PWM_CH(channel)+1,
                    *led_on_p);
```

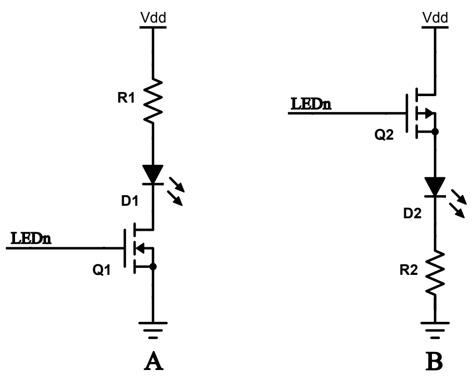
The repository, hosted on Github, is structured in directories with included headers (*/inc*), source files (*/src*) and examples. Each example has its subfolder containing a C++ application and a makefile. Each application makefile is implemented using the *vpath* directive, being able to find the dependencies contained in the include and source folders. The makefile also compiles and links the object files with the g++ compiler, so the user only needs to call the *make* command to get the executable [10].

This framework permits the community to provide new example applications just by implementing a new C++ main code and adding a makefile. This file might be similar to the makefiles earlier implemented.

C. Using the PCA9685 with external drivers

In some applications, just the raw PWM signal may not be driven in the proper way to satisfy the output circuit requirements for voltage or current. It means that some external driving circuit might be necessary.

Figure 7: Simple MOSFET based current driver for LEDs.



For example, servo motors often require a signal on different voltages than that supplied by the SoC, in its 3.3v standard. Or yet, higher power LEDs draw currents greater than the SoC's maximum current per port.

A straightforward answer is to use a driver based on N-MOS or P-MOS transistors. The same circuit could work both as logic level shifter as current driver.

In Figure 7, the MOSFET circuit is used to drive LED current in applications where brightness is controlled by PWM. It can be driven using a N-MOS with the load being pulled-up, or a P-MOS with the load being pulled-down.

Figure 8: Unidirectional logic level shifter circuit.

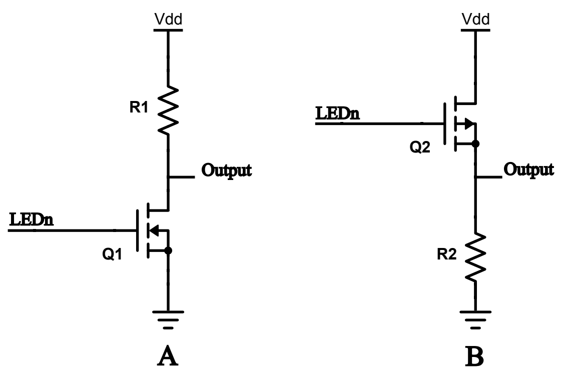


Figure 8 shows the logic level shifter application. For example, a servo motor could be connected to the circuit output to receive angle setpoints via PWM signal.

Depending on the pin output topology configured in MODE2 and the external driver the output signal may be inverted. Also, the PCA9685 has a inverting configuration (bit 4 of MODE2), that enable the inverse logic over the signal. The signal can be foreseen accordingly with the table 1.

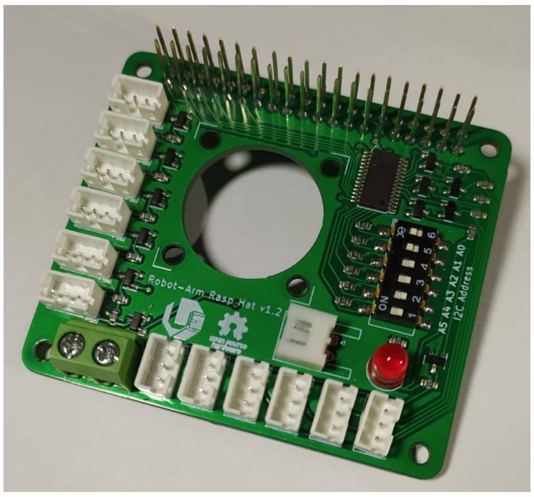
Table 1: Output signal polarity for every output circuit and MODE2 configuration.

External driver	INVRT	OUTDRV	Output signal
A	0	0	output state values inverted
A	0	1	output state values apply
A	1	0	output state values apply
A	1	1	output state values inverted
B	0	0	output state values apply
B	0	1	output state values inverted
B	1	0	output state values inverted
B	1	1	output state values apply

In a study case, a printed circuit board was projected. The board, showed in Figure 9, was made to use the chip with the Raspberry Pi, being able to connect to the single board computer using its pin header. The board adopted the strategy showed in figure 8.A, applying the N-MOS circuit pulled-up. In the projected board, the I^2C address could be easily changed through a DIP switch.

The board's power source have two isolated voltages. The 3.3v voltage, necessary for the chip, is supplied by the Raspberry Pi through the pin header. This smaller voltage is used to energize the chip. The external voltage must be supplied by an external power source connected on the screw terminals. This voltage, varies from 5v to 20v, which is gate-source voltage range for the selected N-MOS (the BSS138) [9] and the possible applications. This power source is used so supply the JST connector that may be wired to servo motors. As well, the external voltage supplies a LED and a cooler fan.

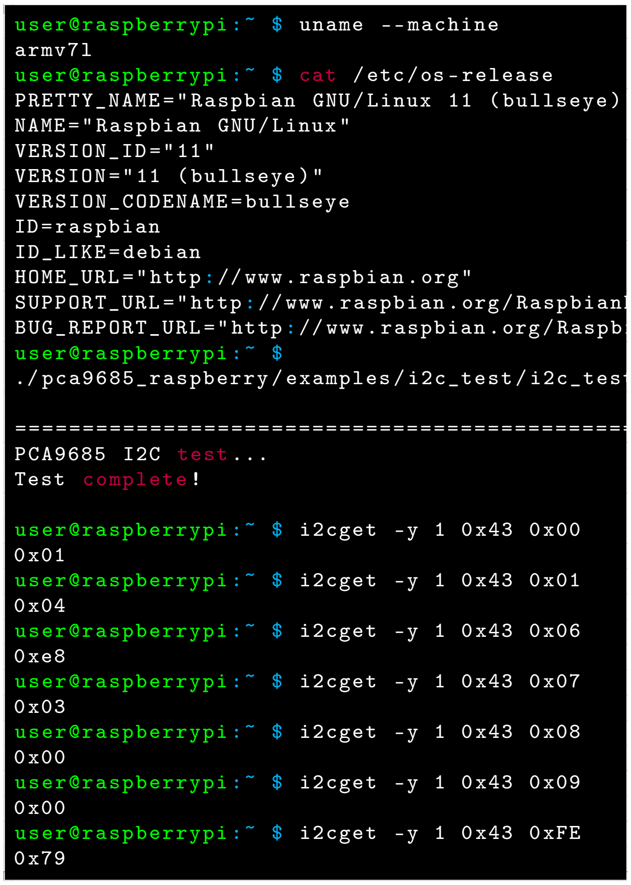
Figure 9: PCA9685 use case. The board was designed to control servo motor with the Raspberry Pi.



III. RESULTS AND DISCUSSION

First of all, it was necessary to test whether the implemented methods correctly modified the PCA9685 registers. To perform this task, a Linux tool called I2C Utilities was used. The tool has a few commands which can be used to accomplish I^2C transactions to slave devices connected to the system [11]. For this test, only the command *i2cget* is used.

Figure 10: Library test on a Raspberry Pi. *I2CUtilities*' commands were used as proof of concept.



```
user@raspberrypi:~ $ uname --machine
armv7l
user@raspberrypi:~ $ cat /etc/os-release
PRETTY_NAME="Raspbian GNU/Linux 11 (bullseye)
NAME="Raspbian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
ID=raspbian
ID_LIKE=debian
HOME_URL="http://www.raspbian.org"
SUPPORT_URL="http://www.raspbian.org/Raspbian
BUG_REPORT_URL="http://www.raspbian.org/Raspb
user@raspberrypi:~ $
./pca9685_raspberry/examples/i2c_test/i2c_tes

=============================================
PCA9685 I2C test...
Test complete!

user@raspberrypi:~ $ i2cget -y 1 0x43 0x00
0x01
user@raspberrypi:~ $ i2cget -y 1 0x43 0x01
0x04
user@raspberrypi:~ $ i2cget -y 1 0x43 0x06
0xe8
user@raspberrypi:~ $ i2cget -y 1 0x43 0x07
0x03
user@raspberrypi:~ $ i2cget -y 1 0x43 0x08
0x00
user@raspberrypi:~ $ i2cget -y 1 0x43 0x09
0x00
user@raspberrypi:~ $ i2cget -y 1 0x43 0xFE
0x79
```

The method adopted was in accordance with the following sequence: a set of commands to read system information, execution of a test code with the library included and a verification using I2C Utilities.

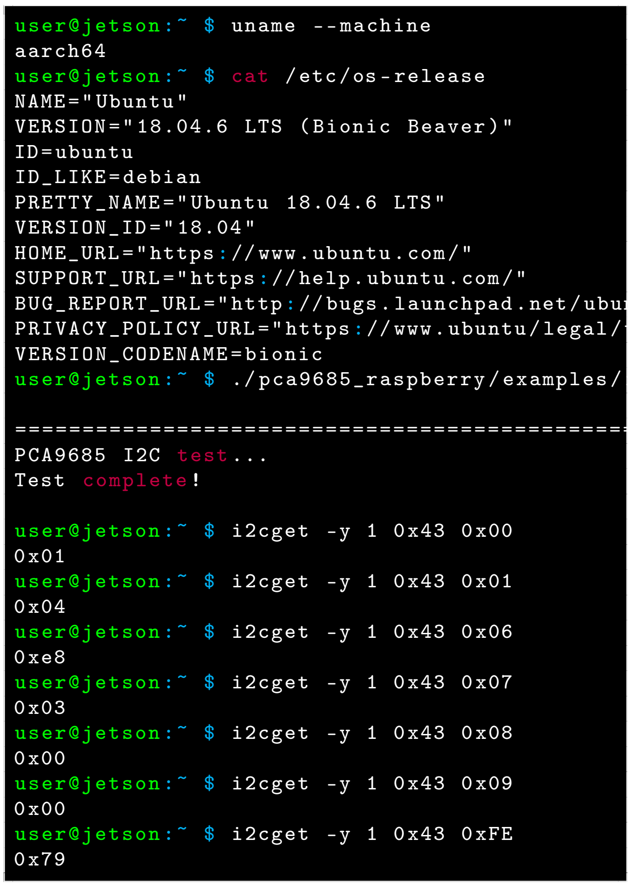
Reading the system info before the test shows the library compatibility across different hardware and operating systems. It was done to find out whether the library is portable or not.

Figure 10 is the terminal view of a test accomplished in the Raspberry Pi. It shows the system have an ARMv7 architecture processor and operates with the Raspbian operating system. Then, the binary *i2c_test* is executed. The code configures MODE1 and MODE2 registers, sets the PWM frequency and writes a duty cycle to the IC's channel 0.

The duty cycle value written to the output is 1000. Checking the values on the registers 0x06 (LED0_ON_L) and 0x07 (LED0_ON_H), it returns the values 0xE8 and 0x03 respectively. As discussed earlier, the 4 LSBs of LED0_ON_H represents the 4 MSBs of the 12-bit duty cycle value. The 8 LSBs after these numbers is the value stored in LED0_ON_L. So, the final value must be formed left shifting 0x03 by 8 and adding 0xE8. Thus, $duty\_cycle = (0x03 << 8) + 0xE8 = 1000$.

The value returned by the last *i2cget* shows the prescale value. The equation 1 can be used to get the frequency, since it is known that $osc\_clock = 25Mhz$ and $prescale\_value = 0x79$. Using these values in the equation, $freq = 50Hz$, as set in the code.

Figure 11: Library test on a NVIDIA Jetson Nano. *I2CUtilities* commands were used as proof of concept..



```
user@jetson:~ $ uname --machine
aarch64
user@jetson:~ $ cat /etc/os-release
NAME="Ubuntu"
VERSION="18.04.6 LTS (Bionic Beaver)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 18.04.6 LTS"
VERSION_ID="18.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubu
PRIVACY_POLICY_URL="https://www.ubuntu/legal/
VERSION_CODENAME=bionic
user@jetson:~ $ ./pca9685_raspberry/examples/

=============================================
PCA9685 I2C test...
Test complete!

user@jetson:~ $ i2cget -y 1 0x43 0x00
0x01
user@jetson:~ $ i2cget -y 1 0x43 0x01
0x04
user@jetson:~ $ i2cget -y 1 0x43 0x06
0xe8
user@jetson:~ $ i2cget -y 1 0x43 0x07
0x03
user@jetson:~ $ i2cget -y 1 0x43 0x08
0x00
user@jetson:~ $ i2cget -y 1 0x43 0x09
0x00
user@jetson:~ $ i2cget -y 1 0x43 0xFE
0x79
```

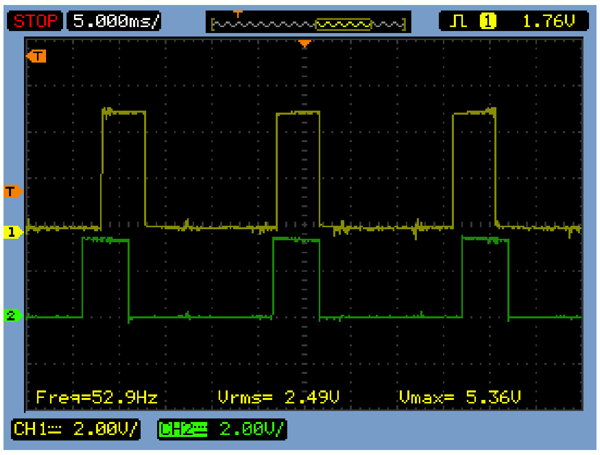
Figure 11 shows the same test performed on the NVIDIA Jetson Nano. The system information says that the system has the aarch64 (64-bit ARM) architecture and the Ubuntu 18.04 operating system. Even with this different environment, the library operated smoothly. It is also possible to check that the I2C Utilities obtained the same values from the chip registers. This suggests that the library has achieved a wide portability.

Once the chip registers was properly modified, it was necessary to verify if the PCA9685 was generating the signal correctly. It was also necessary to check whether the board amplified the signal, as proposed. This task was done with an oscilloscope, which measured the same signal from the projected board and a commercial one. The commercial board does not have the output driver circuit, and so, the voltage level is 3.3v as supplied by the Raspberry.

Figure 12 shows the comparison between the generated signals. The signal in green is that generated with the commercial board. The signal in yellow is that generated with the output drivers, supplied with 5v. It is seen that the output amplification operated successfully.

The Figure 12 also permits checking other output measurements. The measured frequency didn't reaches the value configured in the prescale register accordingly with the equation 1. For the designed board it was measured that the frequency had a value equivalent to $52.9Hz$. This $2.9Hz$ was constant over time. For the commercial board a frequency offset was also observed. In this last case, the measured frequency was $49Hz$, i.e. a $-1Hz$ offset.

Figure 12: Signal amplification analysis using an oscilloscope.



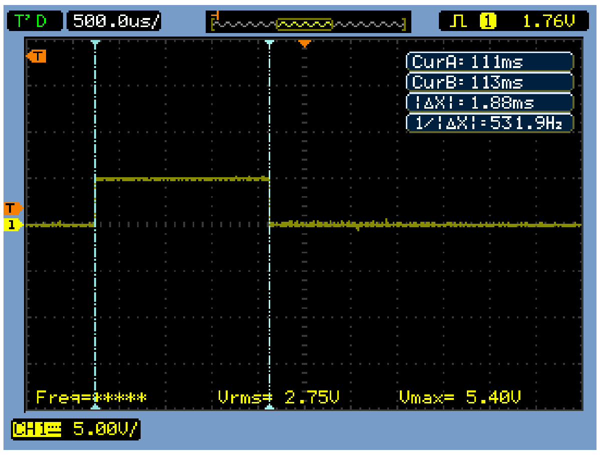
With the signal's frequency known, it was possible to measure the pulse width with the oscilloscope and evaluate if the duty cycle stored in LEDn_ON registers were producing the expected output.

Using the oscilloscope cursors positioned between the voltage pulse, it was measured that the pulse duration was $\Delta x = 1.88ms$. With the ratio between the signal period and the 12-bit value, it is possible to calculate the pulse width. The signal period is the inverse of the frequency value $T = 52.9^{-1} = 18.9ms$. In this example, the integer 408 is an arbitrary duty cycle value. Thus,

$$\frac{18.9}{w} = \frac{4096}{408}$$

which results in $w \approx 1.88ms$. It is approximately the value measured in Figure 13, which proves the accuracy of the generated duty cycle.

Figure 13: Pulse width measuring.



It is possible to correct this frequency bias and get the expected pulse widths. After measuring the offset, it is only necessary to subtract the bias from the desired frequency during the frequency method call.

A second code is present in the library repository, this one was made to test servo motors [7]. In this application, the pulse width must be precisely set to get the expected angle. Therefore, the measured $3.2Hz$ offset was subtracted during the frequency configuration.

Figure 14: PWM with corrected frequency.

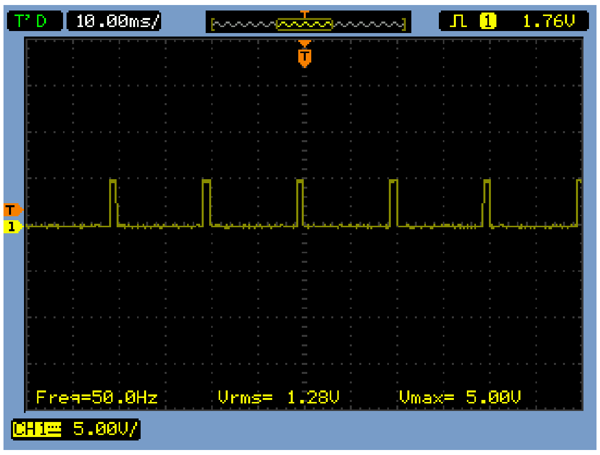


Figure 14 shows that the frequency reached the desired $50Hz$ value. Until now, there's no automation method that automatically corrects the frequency bias.

IV. CONCLUSIONS

The strategy of developing an external hardware for the generation of PWM signals, using the PCA9685, demonstrated that this strategy presents a stable signal output and had with the possibility of integration with embedded applications with Linux as operating system. This absence of jittering permits the solution to be used to control LED intensity or servomotor position with satisfactory stability. However, an offset on the frequency value must be considered. This frequency offset, constant over time, must be estimated before the implementation of an application based on the PCA9685. After this estimation and a proper correction in software, it is possible to implement applications based on the PCA9685's PWM with stability and accuracy.

REFERENCES

[1] ByteSnap. *FreeRTOS vs Linux for Embedded Systems*. Available: `https://www.bytesnap.com/news-blog/freertos-vs-linux-embedded-systems/`. [Accecssed Jul. 2023].

[2] Barkhausen Institut. *Evaluation of PWM Performance of RPi.GPIO and Navio2*. Available: `https://www.barkhauseninstitut.org/research/lab-1/our-blog/pwm-performance`. [Accecssed Jul. 2023].

[3] OLIVEIRA T. (2022). *Entendendo o que é o PWM: a técnica de controle de energia em eletrônica*. Available: `https://eltgeral.com.br/o-que-e-pwm/`

[4] Renesas Electronics Corporation (2019), *The Role of Jitter in Timing Signals*. Available: `https://www.renesas.com/br/en/document/whp/theroleofjitter-intimingsignals`.

[5] R. HIRST (2013), *ServoBlast* [Source code]. Available: `https://github.com/richardghirst/PiBits/tree/master/ServoBlaster`

[6] NXP Semicondutors, *16-channel, 12-bit PWM Fm+ I2C-bus LED controller*, PCA9685 datasheet, Rev. 4, Apr. 2015

[7] V. F. Rodrigues (2023), *pca9685_raspberry* [Source code]. Available: `https://github.com/Sr-Vinicius/pca9685_raspberry`

[8] MOLLOY D. , *Exploring Raspberry Pi: Interfacing to the Real World with Embedded Linux*, John Wiley & Sons, 1st ed., Indianapolis, 2016.

[9] ON Semiconductor, *N-Channel Logic Level Enhancement Mode Field Effect Transistor*, BSS138 datasheet, Rev. 6, Nov. 2021.

[10] Free Software Foundation, *GNU make*. Available: `https://www.gnu.org/software/make/manual/make.html`

[11] RATHORE, S. (2021) *I2C Utilities in Linux*. Available: `https://linuxhint.com/i2c-linux-utilities/`. [Accecssed Jul. 2023].