Military Technical College
Kobry El-Kobbah,
Cairo, Egypt.

18th International Conference
on Applied Mechanics and
Mechanical Engineering.

DEVELOPMENT OF ENGINE-AUTOPILOT INTERFACE CIRCUITS FOR UAVS

M. Yacoub[*]

ABSTRACT

The problem of interfacing Engine Control Units (ECUs) with autopilots in Unmanned Arial Vehicles (UAVs), in the presence of modular communication protocols, is a considerable concern in the UAV industry. This paper presents a development algorithm for interface circuits between ECUs and autopilots for UAVs. The algorithm developed was based on testing and analysing the communication protocols for both the autopilot and the ECU. The engine selected for the analysis was the Zanzottera 498H engine and the autopilot was a generic one that produces PWM commands for the actuators. For off-line testing of the developed interface circuit communication performance with the ECU, a hardware PWM generator circuit was developed to mimic the generated autopilot protocol signal. The present work illustrates the steps to analyse both protocols and the procedures to develop such an interface circuit. Finally, the interface circuit was tested experimentally and showed good performance in communication between the autopilot and the ECU.

KEYWORDS

Unmanned Arial Vehicle (UAV) – Interface Circuit – Autopilot – Engine Control Unit (ECU)

[*] Egyptian Armed Forces.

INTRODUCTION

Before the invention of embedded systems, Unmanned Aerial Vehicles' (UAVs) internal communications included relatively large wired networks represented by complicated harnesses in their design. Since 1992, when a real-time operating system, named "µC/OS; The Real-Time Kernel", was first introduced by Jean Labrosse [1], the embedded systems applications in avionics had evolved. In 1998, the µC/OS-II [2] was introduced and certified in the first commercial avionics product by Federal Aviation Administration (FAA) in 2000 under the DO-178B regulations [3]. Later on, after the development of Controller Area Network (CAN) by Robert Bosch GmbH in the 1980s [4], originally for automotive applications, Boeing and Airbus had developed the ARINC 429 [5] and ARINIC 825 [6] standards for airborne equipment based on the adapted CAN protocols. Although both ARINC 429 and ARINC 825 standards are widely used in avionics till today, as technology advanced, more bandwidths and more flexible topologies were needed. In 2005, ARINC had introduced the ARINC 664-Part 7 standard protocol [7], also referred to as Avionics Full-DupleX Ethernet switching (AFDX). It is an extended standard Ethernet technology with design objectives built around safety.

For the UAV industry, the onboard networks development is underway till the moment of writing this article. Several efforts were recorded to standardise network protocols for UAVs. After CANaerospace was introduced by Stock Flight Systems in 1998 [8] and published by National Aeronautics and Space Agency (NASA) and Advanced General Aviation Transport Experiments (AGATE) in 2001 [9] and refined in 2006 [10], it was used in UAVs in Czech Republic [11] & [12] and Italy [13] & [14].

Despite the benefits of implementing CANaerospace in the embedded systems' networks in UAV platforms, there are some challenges that make these development impractical compared to larger size aeroplanes for the following reasons:

- CANaerospace has a high payload overhead which makes it not suitable for high speed data streams which require low latency.
- Passing multiple values at once in CANaerospace is not an easy task.
- CANaerospace does not provide time synchronization, firmware update and node configuration handling.

As a trial to overcome those challenges, the UAVCAN was introduced [15] and applied in UAV networks [16]. The UAVCAN provides the UAVs with better handling of multiple value passing and lower payload overhead.

As mentioned earlier, onboard networks development in UAVs is underway. Consequently, UAV manufacturers are facing troubles matching their electronic components together, specially, those who produce versatile platforms.

In the present work, an easy approach was proposed to produce an interface circuit to match components in a UAV. The example presented is the Zanzottera 498H engine, Fig. 1, with a generic autopilot.

The approach was based on the determination of the communication protocol of the ECU and the determination of the communication signal produced by the autopilot, which was a PWM signal in the selected autopilot. The target signal to be transferred from the autopilot to the ECU was the desired throttle valve position command signal.

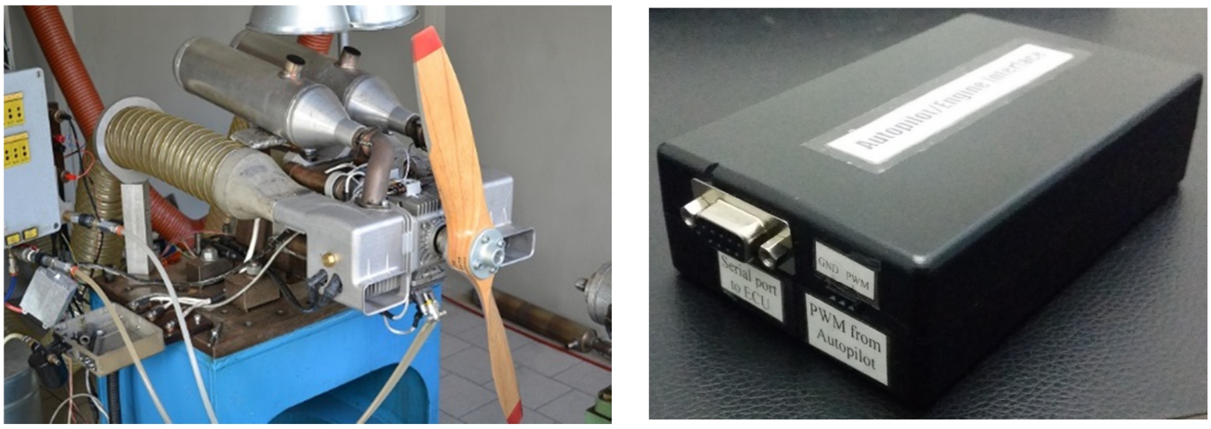

Fig. 1. Zanzottera 498H engine with the developed interface box.

The organization of this article is as follows; the second section contains the signal analysis of the devices, the third sections contains the ECU communication protocol, the fourth section contains the interface circuit design and the algorithm flowchart, the fifth section contains the hardware implementation and testing and, finally, the sixth section concludes the article.

SIGNAL ANALYSIS

In order to propose a design for the interface circuit that handles the communication protocol between two devices, the output and input signals of those devices, the autopilot and the ECU, needed to be analysed. This was accomplished separately for each of the devices. The following analysis illustrates how the signals were measured and analysed for the autopilot and the ECU.

The signal generated by the autopilot for the actuators is a Pulse Width Modulated (PWM) signal. The general waveform of the PWM signal from the autopilot to the throttle valve actuator has the following specifications. The signal has a frequency of 500 Hz and amplitude of 5 V. The duty cycle ranges from 50% to 100% which should correspond to a range from 0% to 100% throttle position. This waveform was implemented originally to target other UAV engines where the throttle valve position control was managed through controlling a stepper motor to push a mechanical lever in order to accelerate the engine. Since the ECU of the engine selected has a different communication protocol as it will described later, the problem of using modular systems arises. To start handling this problem in off-line design stage, a PWM signal generator circuit was developed to mimic the autopilot signal. This PWM signal generator circuit was chosen to be Arduino Uno board. The Arduino program was developed to produce a PWM signal that has the specifications of the autopilot generated PWM signal. Figure 2 shows a sample oscilloscope captured signal for the PWM generated by the Arduino circuit. As shown in figure, the signal follows the specifications of the autopilot PWM signal.

The selected engine, Zanzottera 498H, was equipped with an ECU that receives and transmits signals to the peripheral devices (components of aeroplane) through a serial communication channel. This serial communication channel has a flipped Universal Asynchronous Receive/Transmit (UART) standard with 8-bit data byte, no

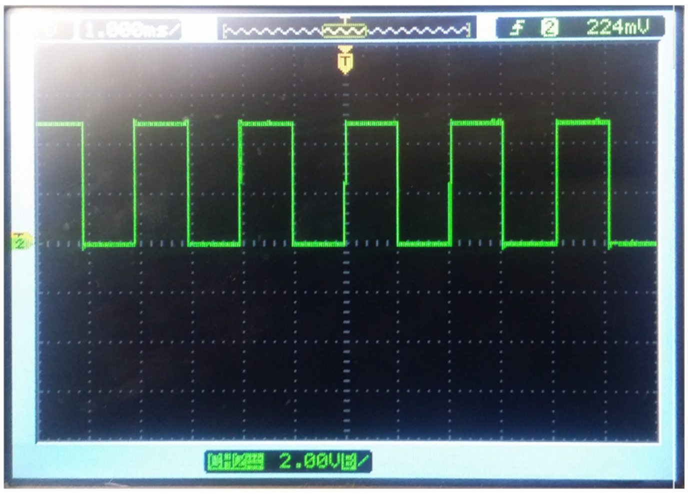

Fig. 2. PWM signal from the autopilot corresponding to the proposed throttle position angle.

parity bit and one stop bit; that is inverse logic UART 8-N-1 standard serial communication. Figure 3 shows a sample signal from the ECU captured on oscilloscope while Figure 4 shows one sample message of the ECU.

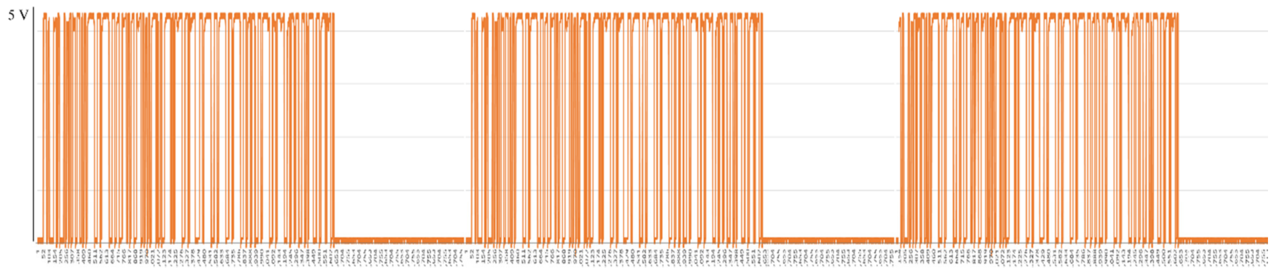

Fig. 3. Three consecutive messages on ECU serial channel captured on oscilloscope.

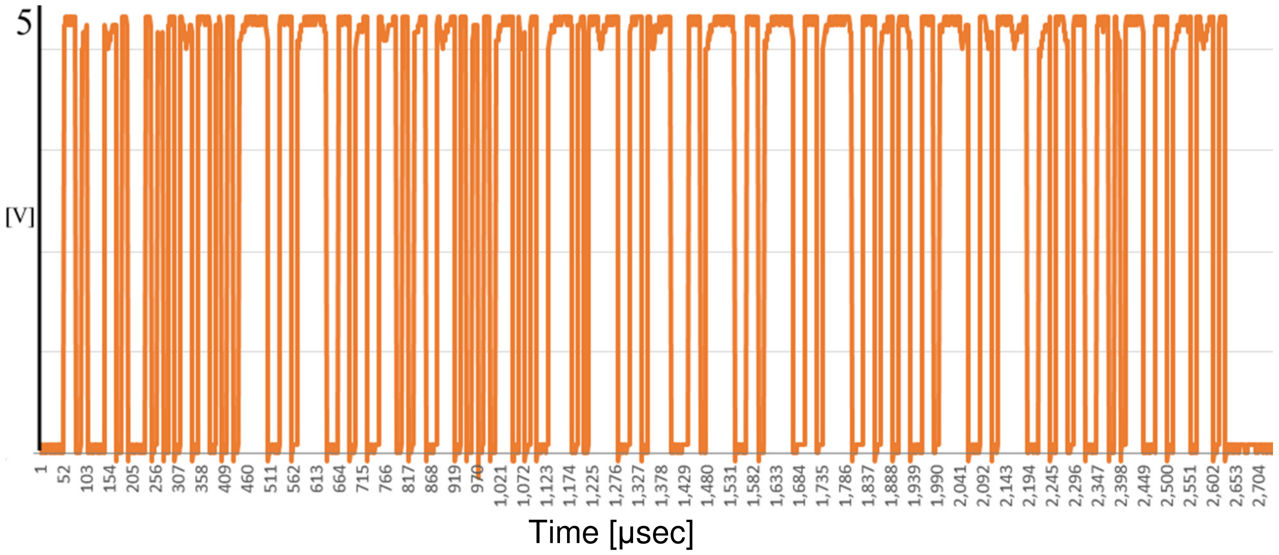

Fig. 4. One sample message from the ECU captured on the oscilloscope.

Analysing messages of the ECU serial communication channel, it is confirmed that the ECU uses the inverse logic UART 8-N-1 standard. The term inverse logic here refers to the low voltage level of the signal idle and the "ones" bits and, conversely, the high voltage level of the "zeros" bits. Notice in Fig. 5 the low voltage of the signal idle and the high voltage level of the start bit. As shown in figure, the first byte of the sample message contains the 10 bits arranged in time order (0-0-1-0-1-1-1-0-0-1). The first bit "0" is the start bit while the last bit "1" is the stop bit according to the UART 8-N-1 standard. The actual byte word is (0-1-0-1-1-1-0-0), where the Least Significant Bit (LSB) is sent first. Hence, the binary number represented by this byte is (00111010)b which is equivalent to 58 in decimal format. This is equivalent to the character "**:**" in the ASCII table.

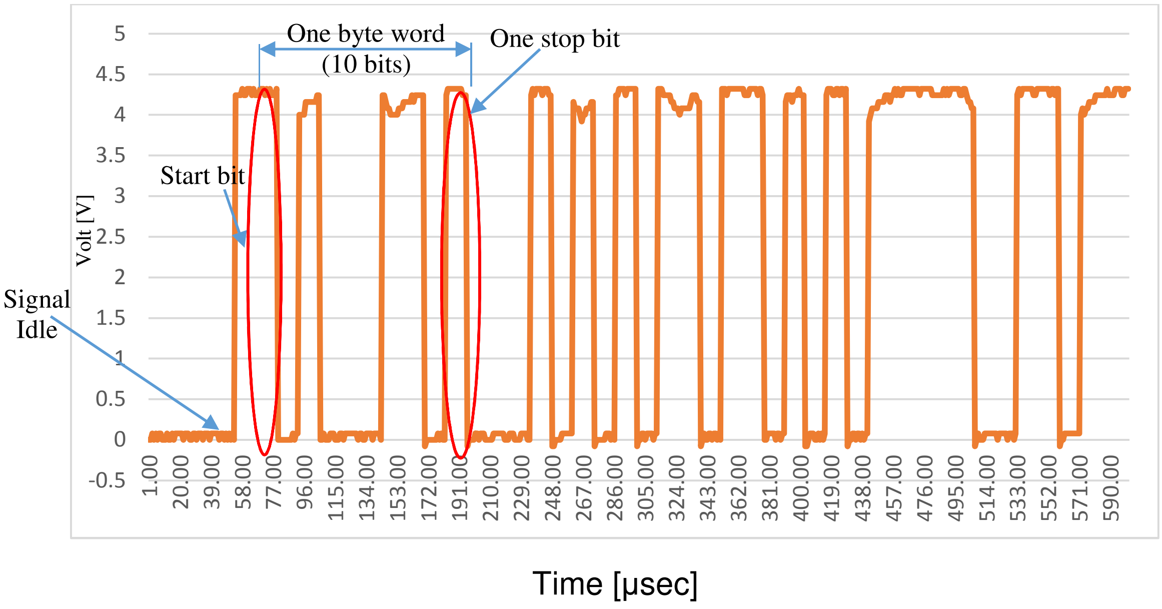

Fig. 5. Sample ECU message captured on oscilloscope; the signal idle is 0V, the start bit is 4.25V, the stop bit is 0V and the total byte word length is 10 bits (0010111001) or (binary 00111010 = decimal 58 = ASCII character "**:**").

ECU COMMUNICATION PROTOCOL

The Zanzottera 498H engine is equipped with an embedded system. This embedded system communicates with the peripherals using a protocol which will be described in this section. Using this protocol, a peripheral device can write to the ECU memory any desired value of the available engine control variables. One of which is the throttle valve position. If such a device wants to write a variable value to the ECU memory, it should send first the write command (three characters "**:WR**") followed by the address of the variable (in hexadecimal format) followed by the byte length of this variable (in hexadecimal format) followed by the value of the variable (in hexadecimal format) and finally followed by the checksum of this whole message (in character format). Notice that the whole message, including the checksum, is sent from the ECU controller in character format which, in turn, is being converted by the UART module in the ECU into its binary equivalent format according to the inverse logic UART 8-N-1 standard described in the previous section. Example complete throttle valve position write message (20 characters): "**WR00FFEC08000201F3[".**

The detailed contents of the above message according to the ECU serial communication protocol is as follows:

- The throttle message starts with write command (three characters) "**:WR**"
- The address of the throttle valve position variable (eight characters in HEX) "**00FFEC08**"
- The data length of the throttle valve position value (four characters in HEX) "**0002**"
- The desired value of the throttle valve position (four characters in HEX) "**01F3**"
- The checksum of the whole message (one character in CHAR format) "**[**"

The checksum used in the ECU is the modulo 256 and is calculated by finding the remainder of the division of the sum of the decimal equivalents (from ASCII table) of all the 19 characters by 256 as follows:

$$\text{Modulo 256 checksum= remainder of } \left(\frac{58+87+82+\cdots+49+70+51}{256}\right) = 91$$

The character equivalent of the decimal value 91 from the ASCII table is the character "**[**".

The checksum character is concatenated to the original 19 characters message to form the complete 20 characters message shown above. This 20 characters message is then sent to the communication channel through the UART one-by-one in 20 consecutive bytes. As mentioned earlier, the UART sends the bytes in 8-N-1 format. This means that each byte is sent in 10 bits (not 8 bits). This is because the UART precedes the original 8 bits with a start bit and ends it with one stop bit without adding any parity bits. Also, this standard sends the 8 bits starting from the LSB first. For example, if the character "**:**" is to be sent, which is equivalent to binary $(00111010)_b$, where the LSB is on the right and the Most Significant Bit (MSB) is on the left, the bit logic will be sent over time as shown in Figure 6.

Refer to Fig. 5 to see the inverse logic of the UART 8-N-1 standard of the character "**:**"while being sent through the ECU serial communication channel.

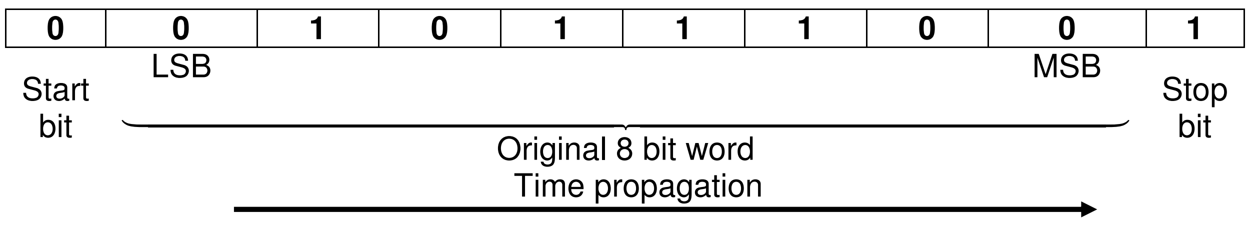

Fig. 6. The bit sequence of the character "**:**" as sent through the communication channel.

INTERFACE CIRCUIT

In order to build an interface circuit to interpret the PWM throttle commands coming from the autopilot and convert them into standard UART serial messages, an Arduino Due board was initially used to do the job. The Due board was chosen due its high processing rate and the multiple serial channels it has which may be used in future development to receive additional signals from the ECU (e.g. engine temperature, engine r.p.m, etc…). As described earlier, another Arduino board was used only in the development phase to simulate the PWM signal coming from the autopilot. The program loaded on the Arduino Due was developed so that after sensing the duty cycle, the program converts this value into a character stuffed inside the character message of the throttle valve position according to the ECU communication protocol.

The Arduino sketch is designed to include two Interrupt Service Routines (ISR); one for detecting the rising of each pulse of the PWM and another for detecting the falling of the same pulse of the PWM signal. To guarantee avoiding nested rising detections, the rising ISR handler was configured in the setup function of the Arduino while the falling ISR handler was configured inside the rising ISR. The pulse width is calculated inside the falling ISR and stored in memory to be used whenever needed in the main loop until it's updated. The main loop conditionally uses the updated pulse duration value every 150 msec. The condition of using the updated value checks if there is a new complete pulse (rise and fall) has arrived by looking at a predefined flag (SW) which is only cleared once a new message has been sent to the serial port. If the condition satisfies, the program calculates the duty cycle and maps it to the range from 0 to 1000 to give an increment of 0.1% throttle opening as recommended by the ECU manual. The actual minimum throttle position accepted by the ECU is 1.8%. Hence, the range was mapped to 26 to 1000.

Since the ECU write command, throttle position value address and throttle position data length are all fixed characters in all messages, they were initially stored in a single CHAR variable in the setup function in the Arduino sketch. Then, the decimal value of the obtained throttle position is converted into HEX and concatenated as characters after the characters of the address and data length. At this point, it is worth mentioning that the Arduino compiler doesn't pad leading zeros of the HEX converted variable. For this reason, a new function was developed to pad leading zeros of the produced HEX variable up to 4 HEX digits (the function is capable of padding zeros up to 8 HEX digits). Then, the checksum is calculated and its CHAR equivalent is concatenated with the original character message. Finally, the complete character message is sent through a serial port. Figure 7 shows the flowchart of program.

HARWARE IMPLEMENTATION & TESTING

This section discusses the hardware implementation and testing of the developed approach. As it will illustrated in this section, the output signal from the Arduino board according to the developed program needed some adaptation before connecting it to the ECU. Also, the developed hardware solution was packed in a subtle fashion to be modular for future development.

Figure 8 shows a sample output message from the Arduino serial port. The Arduino UART produces 8-N-1 standard which has a high (not low) idle level. In addition, the high voltage level is 3.3 V (not 5 V). To solve these two problems, a NOT-gate, e.g. 7404 chip, was used to invert the bit logic of the serial output. To do this, an Arduino shield needed to be designed and built. Both the Arduino Due board and the shield that should hold the 7404 chip were enclosed in one box, namely "Interface Box", Fig. 9 and Fig. 10.

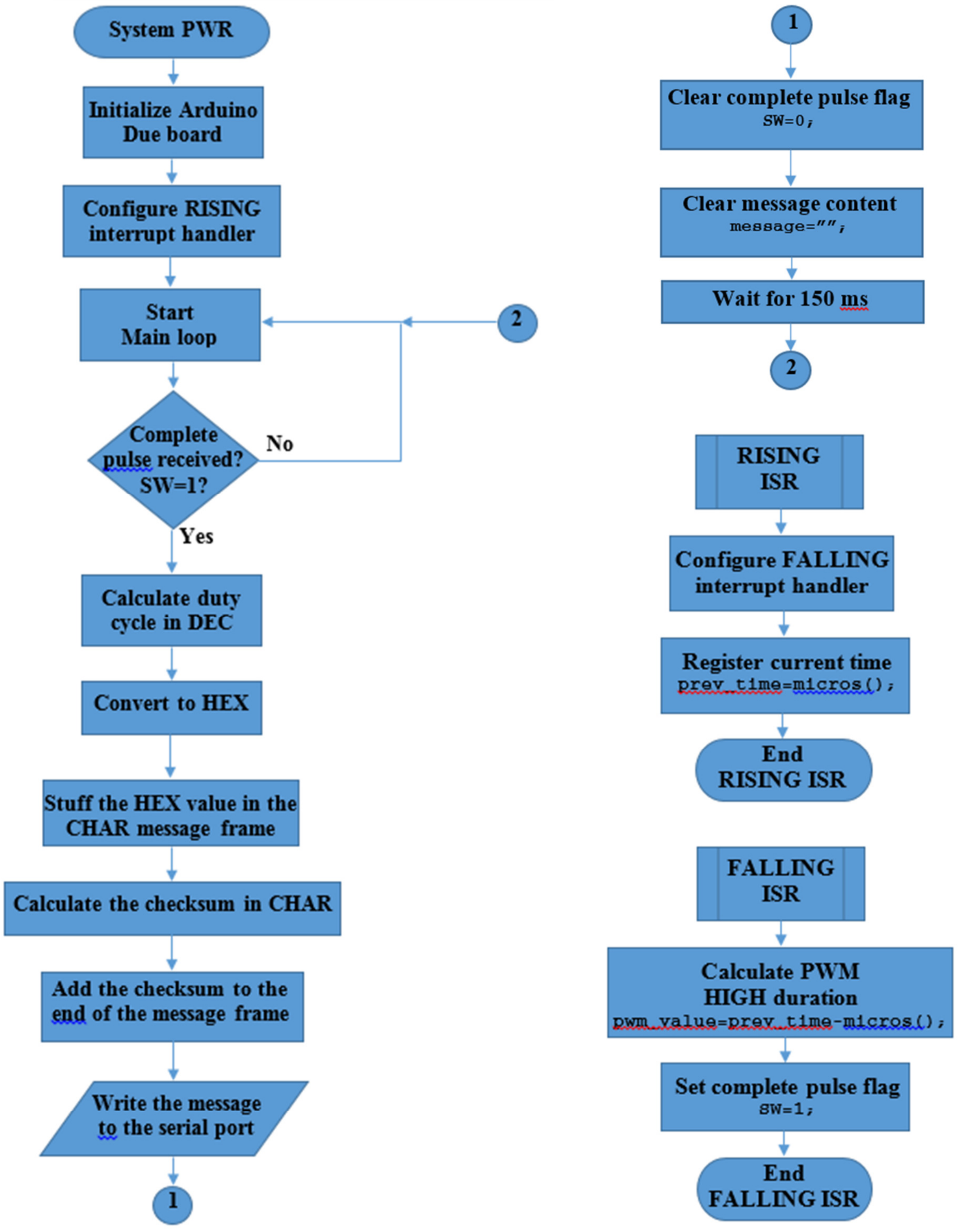
System PWR
Initialize Arduino Due board
Configure RISING interrupt handler
Start Main loop
Complete pulse received? SW=1?
No
Yes
Calculate duty cycle in DEC
Convert to HEX
Stuff the HEX value in the CHAR message frame
Calculate the checksum in CHAR
Add the checksum to the end of the message frame
Write the message to the serial port
1
2
Clear complete pulse flag SW=0;
Clear message content message="";
Wait for 150 ms
RISING ISR
Configure FALLING interrupt handler
Register current time prev_time=micros();
End RISING ISR
FALLING ISR
Calculate PWM HIGH duration pwm_value=prev_time-micros();
Set complete pulse flag SW=1;
End FALLING ISR

Fig. 7. Flowchart of the Arduino program for the interface circuit.

The circuit diagram connecting the Arduino Due board with the NOT-gate chip is shown in Fig. 11. The figure also shows the wiring of the I/O ports of the enclosure of the interface box. As shown on the figure, the interface box has only three ports; power port (+9 V DC), a three-pin servo motor-type port to receive the PWM signal from the autopilot and an RS232 port that transmits and receives serial signals between the interface box and the ECU.

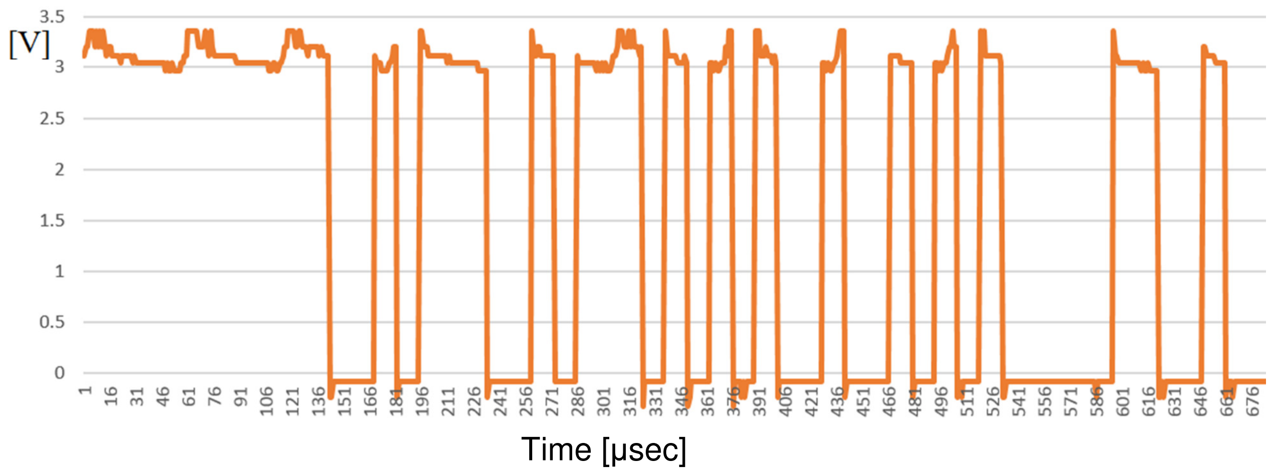

Fig. 8. Output of the Arduino serial port captured on oscilloscope. Notice that the signal idle level at high and the start bit at low.

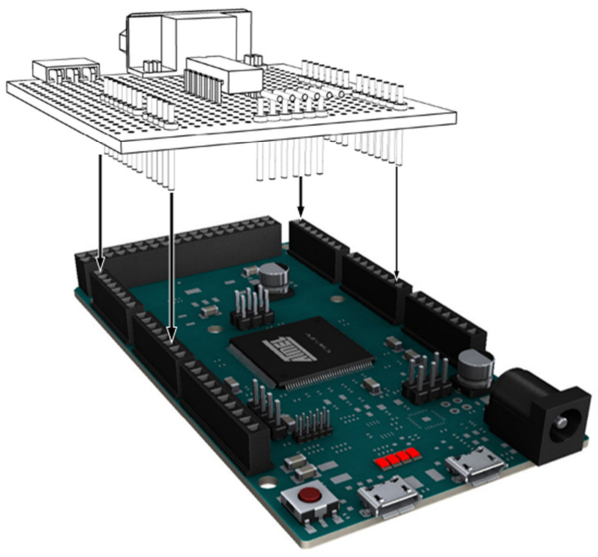

Fig. 9. The mounting of the developed shield on the Arduino Due board.

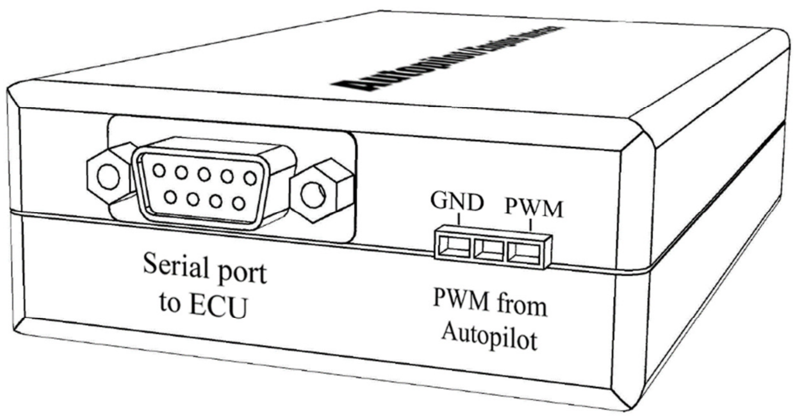

Fig. 10. A model of the developed interface box. Notice the I/O ports of the interface box.

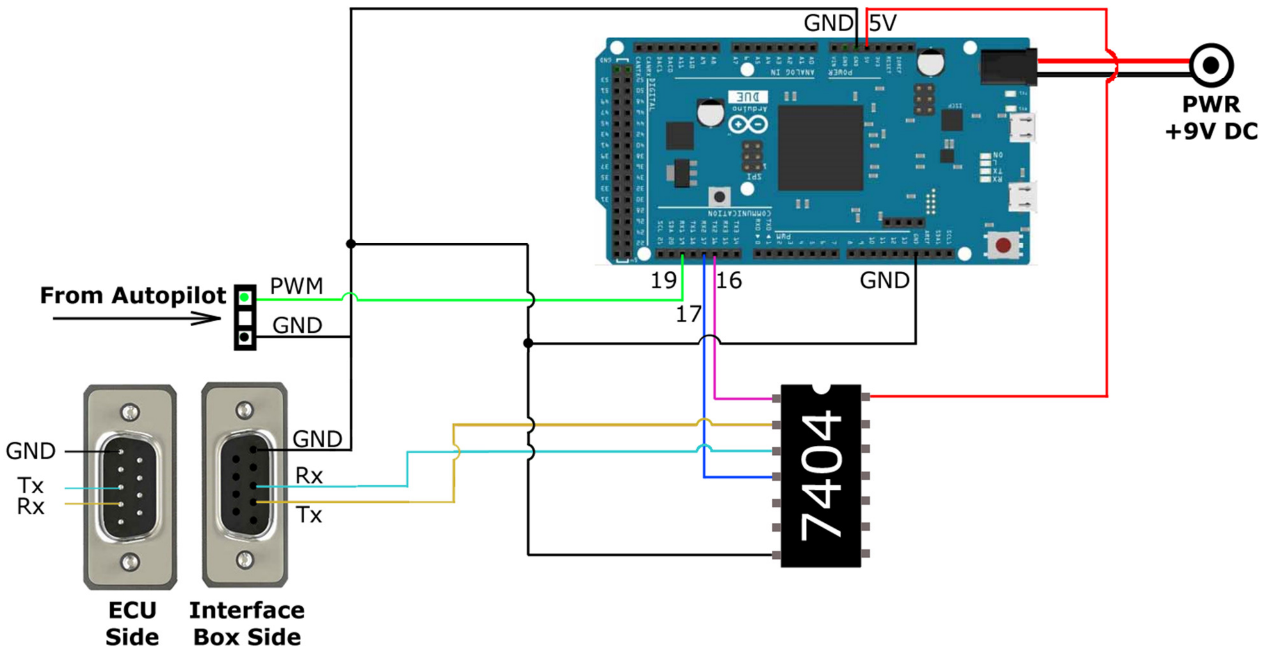

Fig. 11. Circuit diagram showing the connection between the Arduino Due, the NOT-gate chip and the I/O ports from/to the ECU.

Pictures of the developed interface box are shown in Fig. 12. The interface box is designed to be subtle, easy to use and adaptable to the UAV system. The layout of interface box connection with the ECU and the autopilot is shown in Fig. 13. The interface box is connected to the autopilot using a three-pin servo motor-type cable. The interface box is connected to the ECU through a standard RS232 cable. Finally, the interface box is powered by a coaxial power cable from any +9V DC power supply on the UAV. Once connected to the power source, the interface box is powered and activated.

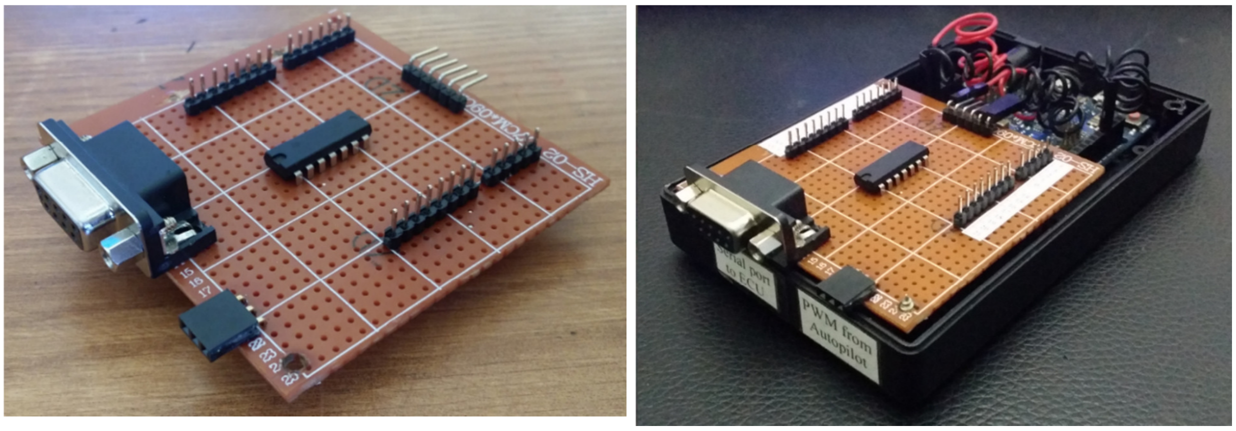

Fig. 12. The developed shield before and after installation on the Arduino Due board.

The developed interface solution was tested on the ECU in a dry run and the output signal was captured on oscilloscope. Figure 14 shows the output signal. As shown in figure, the output signal showed a good match with the required signal by the ECU serial protocol.

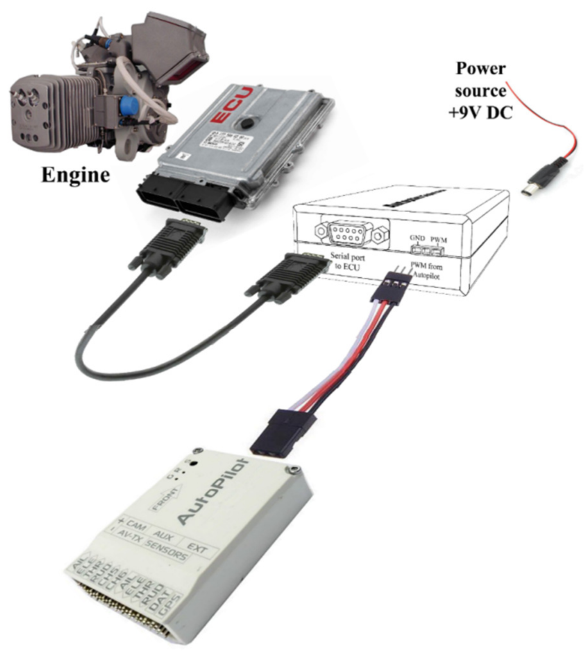

Fig. 13. Interface box connections with the autopilot and the ECU.

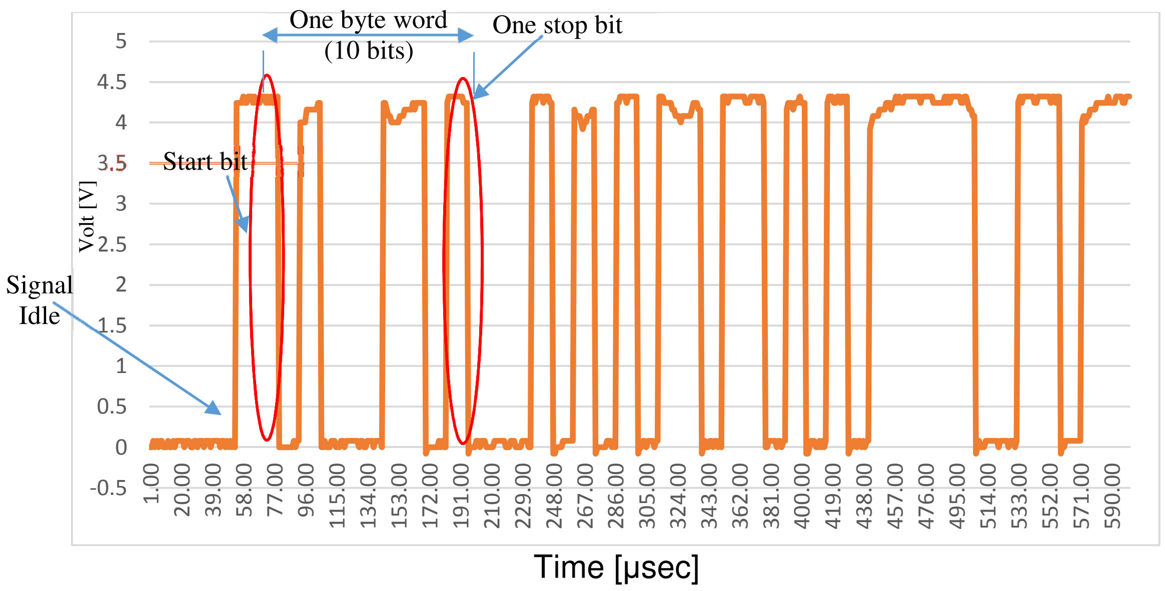

Fig. 14. Sample output message from the interface box captured on oscilloscope; the signal idle is 0V, the start bit is 4.25V, the stop bit is 0V and the total byte word length is 10 bits (0010111001) or (binary 00111010 = decimal 58 = ASCII character "**:**").

The interface box was also tested in the dry run from the point of view of the ECU. The setup included a computer that had a serial monitor to sniff the signal coming from the ECU as a response. The ECU had accepted the signals from the developed interface box successfully. Whenever the computer acquired a read signal for the throttle valve position from the ECU, it was responding with the exact sent commands from the developed interface box.

Finally, the interface box was tested in a live run of the engine. The developed interface box succeeded to enable the control of the engine throttle valve position using a simple joystick. The engine was responsive to every command sent by the joystick through the interface box.

CONCLUSIONS

In the present work, an easy approach to interface two UAV components with two different communication protocols was presented. The approach was based on determining the two communication protocols separately by signal analysis techniques. The example included an ECU and a generic autopilot. An interface circuit was developed to handle the communication between the two components (devices) based on the two communication protocols, the UART 8-N-1 and the PWM. The developed circuit was mounted as a shield for an off-the shelf microcontroller circuit which was loaded with a new developed program illustrated in the present work. Both circuits, the developed shield and the microcontroller board, were enclosed in a single box namely, “Interface Box”. The developed interface box was tested in dry run and live run with the engine. Both tests showed good responses from the engine to the desired PWM signals. The approach presented gives an easy way to aid researches and manufacturers to interface several onboard devices for UAVs in a modular manner.

ACKNOWLEDGEMENTS

This research was fully supported by the Military Technical College. We would like to express our gratitude to the Aircraft Mechanics Department and the Automotive Engineering Department for their support during the course of this research.

REFERENCES

[1] Labrosse, J., “µC/OS: The Real-Time Kernel”, ISBN 978-0-1324-2967-2, CMP Books, December 1992.
[2] Labrosse, J., “MicroC/OS-II: The Real-Time Kernel”, ISBN 978-0-8793-0543-7, CMP Books, 1998.
[3] RTCA Inc., “RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification”, Federal Aviation Administration (FAA), Advisory Circular AC 20-115B, December 1992.
[4] Kiencke, U., Dais, S., Litschel, M., "Automotive Serial Controller Area Network," Society of Automotive Engineers (SAE) International Congress and Exposition, DOI: 10.4271/860391 Detroit, MI, February 1986.
[5] Airlines Electronic Engineering Committee, “ARINC Specification 429P3-18”, Aeronautical Radio, Inc., October 2001.
[6] ARINC 825 Standard, online: https://www.arinc-825.com/arinc825-standard, CAN Aviation Alliance, accessed December 2017.
[7] Airlines Electronic Engineering Committee, "ARINC Specification 664P7", Aeronautical Radio, Inc., June 2005.
[8] Stock, M., “Fly-By-Wire for Experimental Aircraft? A Vision Based on CANaerospace/AGATE Data Bus Technology”,
[9] Langley Research Center National Aeronautics and Space Administration, “System Standard for the AGATE Airplane Avionics Data Bus”, RN. AGATE-WP01-001-DBSTD, October 2001.
[10] M. Stock, "CANaerospace Interface Specification for Airborne CAN Applications V 1.7”, Stock Flight Systems, 2006, available on http://www.stockflightsystems.com, accessed December 2017.

[11] Bajer, J., Bystricky, R., Jalovecky, R., Janu, P., "Aircraft Sensors Signal Processing", RECENT ADVANCES IN MECHATRONICS 2008-2009, pp. 73-78, ISBN: 978-3-642-05021-3, Springer, 2009.

[12] Koukol, O., "Analysis and Certification Method of COTS (Commercial/Cost-Off-The-Shelf) Networks for Aviation Usage", Ph.D. Thesis, University Of Defence Brno, Prague, Czech, 2007.

[13] Catena, A., Melita, C. D., Muscato, G., "An Architecture for Automatic Tuning of the Navigation System of Unmanned Aerial Vehicles", International Conference on Unmanned Aircraft Systems (ICUAS), Atlanta, GA, May 2013.

[14] Catena, A., "Control Architectures for Heterogeneous Fleets of Unmanned Vehicle Systems", PhD thesis, Universita Degli Studi Di Catania, Italy 2014.

[15] P. Kirienko, "UAVCAN", available on http://www.uavcan.org, accessed December 2017.

[16] Silva, P., "Development of Technology and Procedures for Health Monitoring of UAV Subsystems", MSc. thesis, Tecnico Lisboa, Lisbon, Portugal, November 2015.