Simulation of External Peripherals for Automated Testing of Embedded Software in Model MSP430FR5969 [⋆]

José Gustavo de Almeida Machado [*] **Rogério Atem de Carvalho** [**]

[*] *Innovation Hub, Instituto Federal Fluminense, RJ (e-mail: jose.gustavo@gsuite.iff.edu.br).*
[**] *Innovation Hub, Instituto Federal Fluminense, RJ (e-mail: ratem@iff.edu.br)*

Abstract: The use of embedded systems in the various sectors of the economy has grown over the years. Consequently, the demand for software developers capable of creating applications for these systems in an optimized way has increased since the hardware of these devices has limited capacity concerning computers. Embedded software tests are required to validate the correct functioning of its tasks. However, its implementation is not always trivial, as external devices are necessary in order to be able to implement the tests, which may not be available during the project's testing phase. This work aims to present solutions for test software developed for the MSP430FR5969 microcontroller, using ESP32 to simulate the external device used to communicate with the device under test. Test results identify errors found in software developed for the DUT. However, some cases were observed instabilities in the double, reinforcing the need to continue improving the codes used in the double microcontroller. The methodology has the advantage of being software focused, while it is based on any hardware. In addition, it allows the developer greater control over the testing process, quickly making modifications to the double code to explore other scenarios.

Keywords: Embedded Systems; Embedded Software; Automated Driver Testing; ESP32; MSP430.

1. INTRODUCTION

The embedded system is an information processing system that has software and hardware components, with applications in several areas such as telecommunications, automotive, electronics, automation, and military applications. "Some of them include an operating system, but many are so specialized that the entire logic can be implemented as a single program" (Zander, 2008, p. 12).

In order to economically test embedded software, several techniques, approaches, tools and structures have been proposed by professionals and researchers in recent decades (Garousi et al., 2018). However, due to the strong interaction between the drives and the internal and external peripherals of devices, functionality assessment, system performance measurement, and automated testing becomes difficult (Grenning, 2011).

Another challenging factor is the software development for new hardware. A high proportion of hardware defects can be identified during the testing process. In this case, the defects identified may be related to the hardware, not just the software developed (Kualitatem, 2016).

One of the alternatives for testing is for the developer to use real peripherals. However, the hardware may not be available because it is also under development or its high price (Grenning, 2011; Garousi et al., 2018). In addition, "typically the testing team has to share a minimal set of hardware units among its members and/or organize remote access to the hardware" (Garousi et al., 2018, p. 4).

The Model-in-the-Loop is a test technique that consists of performing simulations in a virtual environment, using a mathematical modeling of the system (Plummer, 2006). This technique allows engineers to identify problem requirements.

The Software-in-the-Loop (SIL) technique, on the other hand, aims to compile the source code that will be implemented in the microcontroller and run it in a separate process on the computer, where it performs the simulation (MathWorks, 2021). This feature allows checking the software without having to specify the hardware to be used in the project.

The Hardware-in-the-Loop technique consists of replacing the physical part of a machine or system with a simulation (Kleijn, 2014). In this way, the developer can verify that the software developed for the controller meets specifications without the cost and time associated with actual physical testing. A HIL simulation can demonstrate all the benefits of numerical simulation for other parts of the entire system represented by the software, as long as they can be well modeled (Chen et al., 2008). Some of these

[⋆] This research was supported by the Fundação de Amparo às Pesquisas do Estado do Rio de Janeiro (FAPERJ).

solutions are offered on the market and, depending on the application, can be expensive.

One of the solutions developed to implement tests on embedded systems was using a methodology consisting of three components: a device under test (DUT), a device that simulates the external peripheral (Double) and a personal computer. The DUT contains the code under test and the Double plays the role of the "real" external peripherals that must be used in production. The computer is responsible for executing script tests and compare the results obtained from devices. Therefore, it orchestrates the functions of the microcontroller (Souza and de Carvalho, 2021).

The history of this solution begins with the software to control one of the payloads of the Brazilian 14-BISat nanosatellite. The development of this instrument occurred in parallel to the development of its control software, carried out by the authors, who used basic techniques of Test-Driven Development and test equipment called THC (Carvalho et al., 2014; de Carvalho et al., 2015). These initial experiments generated the Valves 1.0 toolkit, which in turn was used to certify the software embedded in 14-BISat. (Carvalho et al., 2016; de Carvalho et al., 2016).

This paper aims to present test solutions for the software embedded in the MSP430FR5969, using ESP32 as a simulator of the external devices that will connect with the device under test. The applications development are GPS NEO-6M, RTC DS3231 and the transfer of data between devices using the SPI protocol.

2. MATERIALS AND METHODS

Figure 1 shows the architecture used for the tests. Its purpose is to support both iterative and incremental development. It consists of 3 devices: computer, DUT and double.

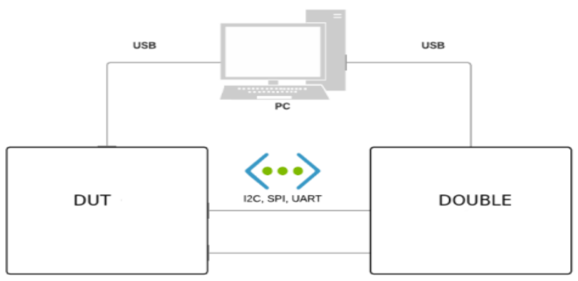


Figure 1. Test System Architecture (Souza and de Carvalho, 2021)

The computer acts as an interface between DUT and Double to control both and perform the automated test. The communication that enables this process is USB/Serial. The computer used in the research contains a 1.7 GHz Intel Core i3 processor, 4 GB DDR3 memory, and the operating system used was Ubuntu 18.04.

The DUT is the device that contains the software that wishes to be tested. In this research, the device used to perform this role is the MSP430FR5969, manufactured by Texas Instruments, where the embedded software was developed in C language.

The double is the device, more specifically a microcontroller, which represents the behavior of one or more real peripherals. Before testing, the application hosted on the computer loads the Double code with one or more precompiled files. The choice of the device to assume the role of double must have a great diversity of protocols and peripherals, and an affordable price (Souza and de Carvalho, 2021). In this work, the device used to perform the double role is the ESP32, with the standard CPU frequency of 160 MHz.

The test cases verify the response of the device with the expected result after injecting input data. Due to the different types of language paradigms used in the development of DUT and double software, since the C language belongs to the imperative paradigm and the Python language belongs to the object-oriented paradigm, the data entry for each device must be performed differently.

For each test case, the following steps are included:

(1) Initialization: consists of the initialization of an object in the Double by the test application. In the case of DUT, the production code is already initialized, regardless of the test code.
(2) Input injection: consists of positioning the production code in a state to perform a testing activity. The test application is also responsible for this positioning. Data input for the MSP430FR5969 varies depending on the issue. Besides that, the MSP430 will interpret and process this data according to its activity in question. For example, entering a set of numbers, such as a list, must use the following command "-y 1 2 3". In other cases, sending a flag, such as a letter "y", is enough to perform some activity, simulating, for example, a button.
(3) Results Gathering: consists of requesting and collecting results from the DUT or the Double through the test. Obtaining the results in the Double is done by calling a method by its object. The results are obtained in the DUT by command, as shown in the previous step.
(4) Asserts: it consists of comparing the result obtained with the expected result.

This methodology is a choice over other given the following reasons:

(1) Different the Model in the Loop and Software in the Loop techniques, the signals sent to the device under test are generated by hardware. This allow to verify the response of the microcontroller as if it were in a production environment.
(2) It is a low-cost technique compared to other tools that use the Hardware-in-the-Loop concept. Having the code of the Double, it is possible to use a microcontroller that is not in use to be able to act in the tests.
(3) The software needed for microcontrollers to play the role of Double can be developed by the community and reused in multiple projects, saving time and costs.

The methodology has the advantage of being software focused, while it is based on any hardware. In addition, it allows the developer greater control over the testing process, quickly making modifications to the double code to explore other scenarios.

The codes used for the Double and the test scripts base were developed by Sara Monteiro, which were made available to the public through the GitHub repository entitled Micropyhon Test Lib (Souza, 2020). The reuse of these codes is interesting because it allows validating these algorithms developed using other models of embedded systems, ensuring reliability and encouraging the community to use and improve them.

The production codes are the objects to be tested. The script tests implemented for MSP430 need to send commands and data via the terminal to execute a function. For this reason, a module that works with the UART protocol must be dedicated to the tests. The author of this research developed them.

The test codes, DUT and Double, used in this paper, were published in the GitHub repository to collaborate with the community and improve the technique (Machado, 2021).

3. RESULTS AND DISCUSSIONS

3.1 UART - GPS

This application makes use of an external GPS module model NEO-6M using the UART protocol. The GPS is configured through commands that indicate the messages types it must report, known as NMEA sentences (National Marine Electronics Association) (Souza and de Carvalho, 2021).

The first test case consists of sending a command to the GPS. The command used enables the GGA and RMC sentences on the GPS. The command sent to the DUT is the "y" flag, which triggers the function that performs this activity.

The second test case checks the GPS response after sending a command through the DUT. The Double evaluates the rate update command from the command sent and, if positive, internally calls the continuous_mode() method, enabling a timer interrupt to send a Double sentence RMC periodically. The sentence received by the DUT is verified and the command to obtain this information is the "b" flag.

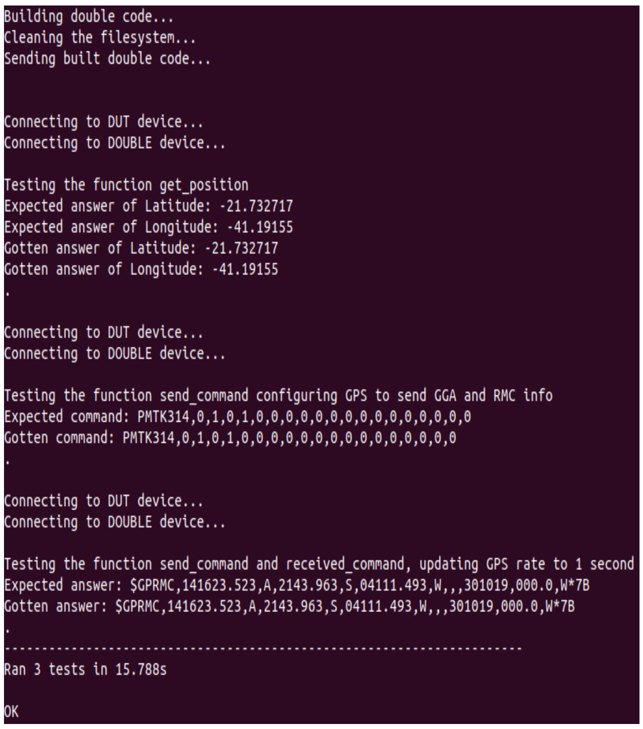
The third test case checks the latitude and longitude value that the DUT received from the GPS after the request. The command to obtain this information from the DUT is "u".

Code 1 presents the first test case, which verifies that the command sent by the DUT is correctly received by the GPS (Double).

```
expected_command = "PMTK314
    ,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0"
# 1 - Objects Creation
self.double_serial.repl("gps =
    DOUBLE_GPS(22,16,2)", 0.2)

# 2 - Input Injection
self.dut_serial.repl("y", 1.2)

# 3 - Results gathering
gotten_command = self.double_serial.
    repl("gps.received_command()", 0.8)
    [2]

# 4 - Assertion
gotten_command = gotten_command.decode
    ()
gotten_command |should| equal_to (
    expected_command)
```

Code 1. Test case for the function of sending the command to the GPS

Figure 2 shows the result of the executed test.



```
Building double code...
Cleaning the filesystem...
Sending built double code...


Connecting to DUT device...
Connecting to DOUBLE device...

Testing the function get_position
Expected answer of Latitude: -21.732717
Expected answer of Longitude: -41.19155
Gotten answer of Latitude: -21.732717
Gotten answer of Longitude: -41.19155
.

Connecting to DUT device...
Connecting to DOUBLE device...

Testing the function send_command configuring GPS to send GGA and RMC info
Expected command: PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
Gotten command: PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
.

Connecting to DUT device...
Connecting to DOUBLE device...

Testing the function send_command and received_command, updating GPS rate to 1 second
Expected answer: $GPRMC,141623.523,A,2143.963,S,04111.493,W,,,301019,000.0,W*7B
Gotten answer: $GPRMC,141623.523,A,2143.963,S,04111.493,W,,,301019,000.0,W*7B
.
----------------------------------------------------------------------
Ran 3 tests in 15.788s

OK
```

Figure 2. Test result for GPS after execution

3.2 I2C - RTC DS3231

The tested application is the system that configures an external RTC of the DS3231 model, using the I2C communication protocol. The activities that will be tested are configuring and reading the date and time in the RTC. In all, four test cases were developed.

The firmware used in ESP32 was "MicroPython LoBo esp32", with the date of the last update on September 4, 2018. The manufacturer of ESP32 did not develop this firmware, but its use is necessary because the original firmware does not support the protocol I2C in slave mode. Furthermore, there was a need to configure the firmware to work with the CPU with a default frequency of 160 MHz since the precompiled models, available for downloads, are configured for 240 MHz.

As shown in Code 2, the first test case was defined to test the date and time configuration. This test is static as the date and time values stored by the Double do not change over time. Therefore allows the developer to verify that the function that performs this activity is storing the values accurately. Besides that, it is unnecessary to use the function that captures the date and time on the external device.

```
global double_serial
expected_datetime = [2021, 5, 22, 6,
    20, 14, 31]

# 1 - Objects Creation
double_serial.repl("ds3231 =
    DOUBLE_DS3231(21,22)", 0.5)

# 2 - Input Injection
self.dut_serial.repl("-t 0 31 14 20 6
    22 5 21z", 1)

# 3 - Results gathering
gotten_datetime = double_serial.repl("
    ds3231.DateTime()", 1.2)

# 4 - Assertion
gotten_datetime = gotten_datetime[3].
    decode()
gotten_datetime | should | equal_to(
    str(expected_datetime))
```

Code 2. Test case for sending data function to configure RTC

The DUT will transmit a set of data necessary for configuring the RTC. For example, the command "-t 0 31 14 20 6 22 5 21z" can be interpreted as follows:

(1) The "-t" flag informs the desired activity is to configure the date and time in Double.
(2) The value "0" indicates the address of the first register that will store the information received. This register has the function of storing seconds.
(3) The remaining integer values are seconds, minutes, hour, day of the week, day, month, and year's last two digits.
(4) The "z" flag indicates the end of the command.

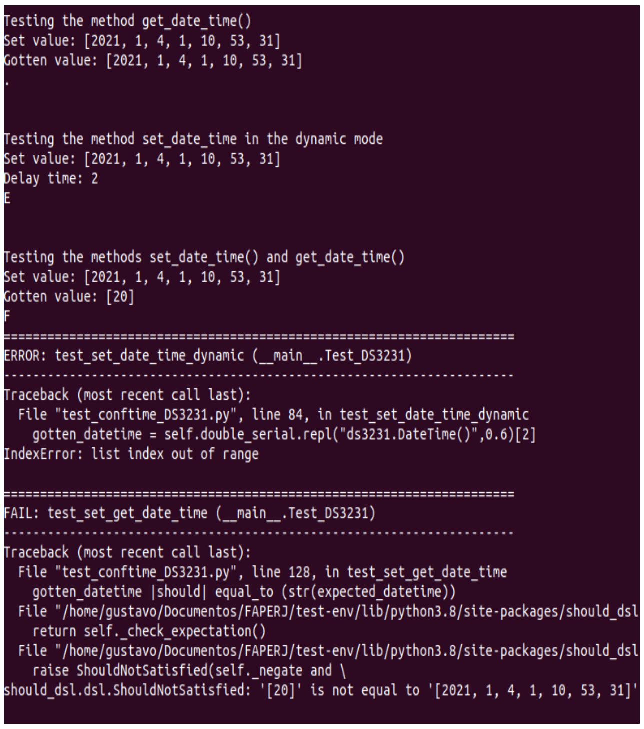
The second test case, whose function is similar to the previous case, except that it works dynamically, i.e., the values stored by the Double registers will be updated every 1 second. It is necessary to check the dynamic situation closer to reality.

The third test case checks the function that requests the RTC date and time. This case does not use the function that sets the date and time, bringing independence between the tested codes. This fact is possible thanks to the possibility to configure the Double through the test application directly.

The latter case tests the date and time configuration and request functions together. It is helpful to perform a self-test, i.e., the device to be tested is responsible for providing the input and output data for verification.

For the correct execution of the test script, there was a modification to the standard used on test codes obtained in project Micropython Test Lib. During test runs, errors were usually raised after the first successful test case was run. In the problem investigation, it was found that the test cases execution in isolation and without the need for more than one opening in the connection of the Double with the computer obtained the expected results. However, the error is generated in opening and closing connection several times between Double and computer because the MicroPython REPL prompt fails and the device does not receive the commands sent by the test script. The solution was open and close communication between Double and the computer one time to avoid REPL failure.

Figure 3 shows the test case errors for applying the RTC configuration. Besides the possibility of DUT code error, there is the possibility that the functioning of Double causes some failures. Another possibility is the delay in communication between the computer used for the test and the devices.



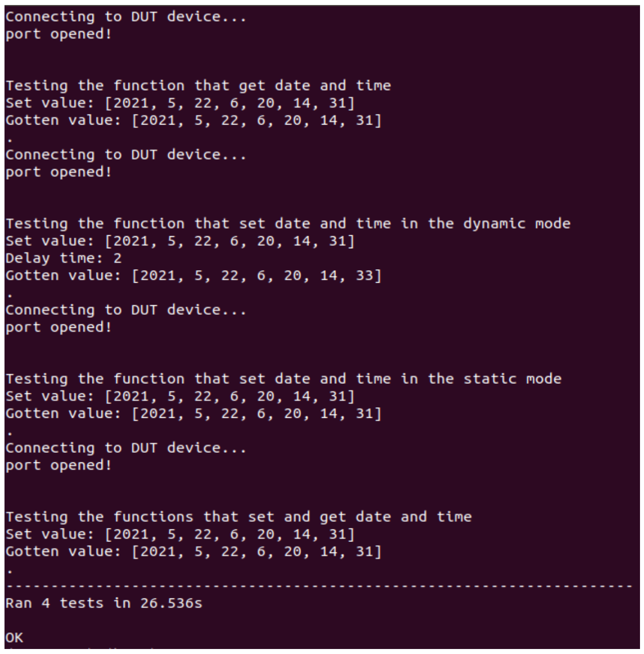
```
Testing the method get_date_time()
Set value: [2021, 1, 4, 1, 10, 53, 31]
Gotten value: [2021, 1, 4, 1, 10, 53, 31]
.


Testing the method set_date_time in the dynamic mode
Set value: [2021, 1, 4, 1, 10, 53, 31]
Delay time: 2
E


Testing the methods set_date_time() and get_date_time()
Set value: [2021, 1, 4, 1, 10, 53, 31]
Gotten value: [20]
F
======================================================================
ERROR: test_set_date_time_dynamic (__main__.Test_DS3231)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_conftime_DS3231.py", line 84, in test_set_date_time_dynamic
    gotten_datetime = self.double_serial.repl("ds3231.DateTime()",0.6)[2]
IndexError: list index out of range

======================================================================
FAIL: test_set_get_date_time (__main__.Test_DS3231)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_conftime_DS3231.py", line 128, in test_set_get_date_time
    gotten_datetime |should| equal_to (str(expected_datetime))
  File "/home/gustavo/Documentos/FAPERJ/test-env/lib/python3.8/site-packages/should_dsl
    return self._check_expectation()
  File "/home/gustavo/Documentos/FAPERJ/test-env/lib/python3.8/site-packages/should_dsl
    raise ShouldNotSatisfied(self._negate and \
should_dsl.dsl.ShouldNotSatisfied: '[20]' is not equal to '[2021, 1, 4, 1, 10, 53, 31]'
```

Figure 3. RTC application test results with the error generated by the fail of the connection between the Double with computer

Figure 4 shows the execution of the tests with positive results. The difference between this test result and the previous one is the opening connection only one time between the computer and Double during the entire test run. A global variable is created, which stores the instance of the SerialInterface class responsible for connecting to devices.

3.3 SPI - Master-slave

The test cases for this application consist of the transfer of data between the DUT with Double using the SPI protocol. The first case verifies a read operation on the slave by the master. The Double, for this example, plays the role of an external peripheral typically performed by ADCs.



```
Connecting to DUT device...
port opened!


Testing the function that get date and time
Set value: [2021, 5, 22, 6, 20, 14, 31]
Gotten value: [2021, 5, 22, 6, 20, 14, 31]
.
Connecting to DUT device...
port opened!


Testing the function that set date and time in the dynamic mode
Set value: [2021, 5, 22, 6, 20, 14, 31]
Delay time: 2
Gotten value: [2021, 5, 22, 6, 20, 14, 33]
.
Connecting to DUT device...
port opened!


Testing the function that set date and time in the static mode
Set value: [2021, 5, 22, 6, 20, 14, 31]
Gotten value: [2021, 5, 22, 6, 20, 14, 31]
.
Connecting to DUT device...
port opened!


Testing the functions that set and get date and time
Set value: [2021, 5, 22, 6, 20, 14, 31]
Gotten value: [2021, 5, 22, 6, 20, 14, 31]
.
----------------------------------------------------------------------
Ran 4 tests in 26.536s

OK
```

Figure 4. RTC application test results after solving the connection error between the Double device and computer

Code 3 presents an example of a command for the DUT to perform the activity related to that test is ”-g 3z”, where the flag ”-g” means that the desired function, the value ”3” means the number of data to be read and the ”z” flag means the end of the command.

```
global double_serial
expected_readings = [1,2,3]

# 1 - Objects Creation
double_serial.repl("slave =
    Double_slave(13,12,14,15)",0.5)

# 2 - Input Injection
double_serial.repl("slave.
    enable_transaction("+str(
    expected_readings)+")",0.5)

# 3 - Results gathering
gotten_readings = self.dut_serial.repl
    ("-g 3z", 1.2)[0]
# ...

# 4 - Assertion
# ...
str(gotten_readings) |should| equal_to
     (str(expected_readings))
```

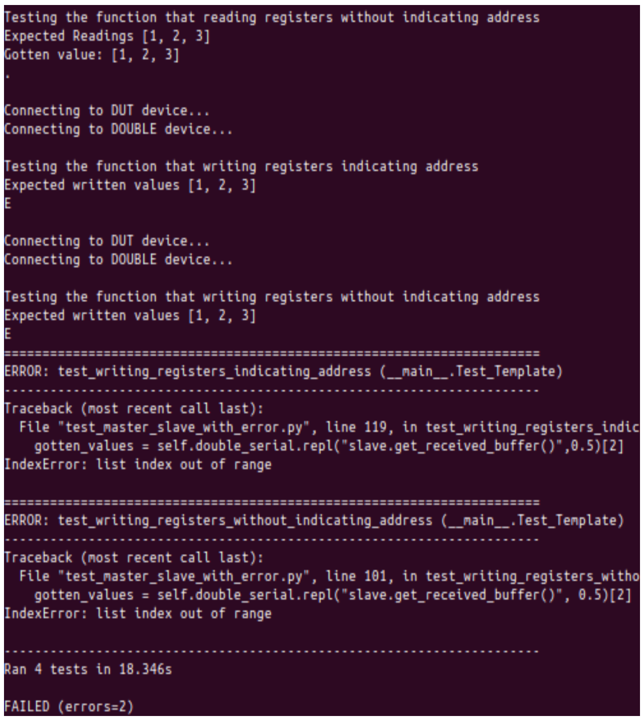
Code 3. The DUT function test that reads data from Double

The second test case is similar to the first, except that the DUT reports the buffer address. An example of a command for this activity is ”-a 14 3z”, where the flag ”-a” means the desired function, the value ”14” is the buffer address, the value ”3” indicates the number of data to be received, and the ”z” flag means the end of the command.

The third test case checks whether the Double records the data reported by the DUT. An example of a command is the ”-s 1 2 3z”, where the ”-s” flag means the desired function, the numeric data is the data to be written, and the ”z” flag is the end of the command.

The fourth test case checks whether the Double records the data sent by the DUT at the address provided. An example of a command is ”-s 14 1 2 3z”, which the only difference between the last command is the address information indicated by the value ”14”.

Figure 5 shows application test result errors. The API needed to use the SPI protocol in slave mode on ESP32 devices was developed by Sara Monteiro, which was used in Double (Souza, 2020). However, according to the developer, the API is unstable (Souza and de Carvalho, 2021). Therefore, it can cause errors in the test script execution, although the functions developed for the DUT are correct.



```
Testing the function that reading registers without indicating address
Expected Readings [1, 2, 3]
Gotten value: [1, 2, 3]
.

Connecting to DUT device...
Connecting to DOUBLE device...

Testing the function that writing registers indicating address
Expected written values [1, 2, 3]
E

Connecting to DUT device...
Connecting to DOUBLE device...

Testing the function that writing registers without indicating address
Expected written values [1, 2, 3]
E
======================================================================
ERROR: test_writing_registers_indicating_address (__main__.Test_Template)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_master_slave_with_error.py", line 119, in test_writing_registers_indic
    gotten_values = self.double_serial.repl("slave.get_received_buffer()",0.5)[2]
IndexError: list index out of range

======================================================================
ERROR: test_writing_registers_without_indicating_address (__main__.Test_Template)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_master_slave_with_error.py", line 101, in test_writing_registers_witho
    gotten_values = self.double_serial.repl("slave.get_received_buffer()", 0.5)[2]
IndexError: list index out of range

----------------------------------------------------------------------
Ran 4 tests in 18.346s

FAILED (errors=2)
```

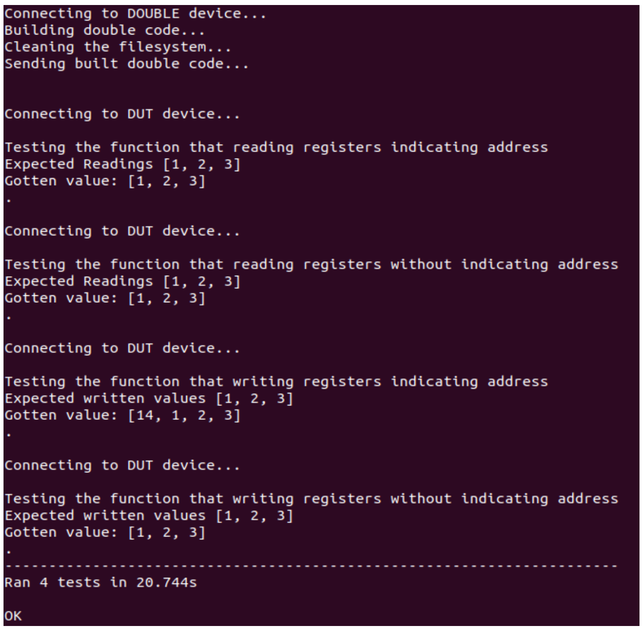
Figure 5. Test execution of the Master and Slave application with error

The same strategy used in the RTC tests was adopted to increase the test reliability, which is the only opening of the connection with the Double by the computer during the whole execution of the test. This action allowed the errors generated to be restricted to the results generated by the embedded software, which is the objective of this work.

Figure 6 shows the test result using the strategy described above. It allowed for reliability. However, in some cases, results similar to what was reported by the API developer (Souza and de Carvalho, 2021) were observed.

4. CONCLUSIONS

The use of external peripheral simulators for testing software in embedded systems proves valuable because it



```
Connecting to DOUBLE device...
Building double code...
Cleaning the filesystem...
Sending built double code...


Connecting to DUT device...

Testing the function that reading registers indicating address
Expected Readings [1, 2, 3]
Gotten value: [1, 2, 3]
.

Connecting to DUT device...

Testing the function that reading registers without indicating address
Expected Readings [1, 2, 3]
Gotten value: [1, 2, 3]
.

Connecting to DUT device...

Testing the function that writing registers indicating address
Expected written values [1, 2, 3]
Gotten value: [14, 1, 2, 3]
.

Connecting to DUT device...

Testing the function that writing registers without indicating address
Expected written values [1, 2, 3]
Gotten value: [1, 2, 3]
.
----------------------------------------------------------------------
Ran 4 tests in 20.744s

OK
```

Figure 6. Test execution of the Master and Slave application successfully

validates the elaborated codes and visualizes errors that would go unnoticed by developers. It can be a low-cost alternative, as there is no need to purchase the equipment to start development and tests.

This model also allows a more significant number of developers to work in parallel. Each can work on a software module and test it without having the external peripheral available because Double codes can be used on idle devices.

Another feature of this test model is that it does not use breakpoints and a special debug mode to run the tests, but there is a delay between sending commands and receiving responses through the serial port.

Differing from the GPS tests, which use the UART protocol for communication, the RTC tests and data transfer between devices on the SPI bus showed errors not related to the algorithms developed for MSP430FR5969 but between the computer's communication with the Double. This situation may be related to the fact that the manufacturer of ESP32 did not develop the firmware used in Doubles for these tests since the UART protocol, used in Double in the case of GPS, has the functioning module as a slave by the original firmware. Another possibility of the problem is the delay between the computer's communication with the Double.

The solution was to create the communication between the computer and Double only once during the entire test, which does not impact the model proposal, as all steps are maintained. Communication between computers and devices is used only for tests, unlike the communication between the devices, which will be used in production. Therefore, the main objective of the test is fulfilled.

Contrasting the Double, the communication opening and closing between the computer and DUT is performed several times during the test execution. This fact demonstrates more stability in the communication between the computer and MSP430FR5969 than between the computer and ESP32. For future work, built-in software for MSP430FR5969 will be developed to function as a Double for tests.

ACKNOWLEDGMENTS

This work is supported by Fundação de Amparo às Pesquisas do Estado do Rio de Janeiro (FAPERJ) and Pólo de Inovação de Campos dos Goytacazes.

REFERENCES

Carvalho, R., Arueira, G., Azevedo, M., and Toledo, R. (2016). Developing the software for cubesats in a concurrent engineering environment: a toolset and case study. In *Second IAA Latin American CubeSat Workshop IAA Latin American CubeSat Workshop*.

Carvalho, R., Ferreira, H., Toledo, R., Cordeiro, C., and Moura, G. (2014). Interfacing with the science unit: Preparing the software side. In *Proceedings of the 6th European CubeSat Symposium*.

Chen, X., Salem, M., Das, T., and Chen, X. (2008). Real time software-in-the-loop simulation for control performance validation. *Simulation*, 84, 457–471. doi: 10.1177/0037549708097420.

de Carvalho, R.A., de Azevedo, M.S., de Souza, S.C., Arueira, G.V., and Cordeiro, C.S. (2016). Developing and testing software for the 14-bisat nanosatellite. *IFAC-PapersOnLine*, 49(30), 71–74.

de Carvalho, R.A., Silva, H., Toledo, R.F., and de Azevedo, M.S (2015). Tdd for embedded systems: A basic approach and toolset. *arXiv preprint arXiv:1507.07969*.

Garousi, V., Felderer, M., Karapıçak, Ç.M., and Yılmaz, U. (2018). Testing embedded software: A survey of the literature. *Information and Software Technology*, 104, 14–45.

Grenning, J.W. (2011). *Test Driven Development for Embedded C*. Pragmatic bookshelf.

Kleijn, C. (2014). Introduction to hardware-in-the-loop simulation.

Kualitatem (2016). Embedded software testing challenges. URL `https://www.kualitatem.com/blog/embedded-software-testing-challenges`.

Machado, J.G.A. (2021). msp430-test-with-esp32. URL `https://github.com/machado-jose/msp430-test-with-esp32`.

MathWorks (2021). Software-in-the-loop simulation. URL `http://wwtug.org/instmem.html`.

Plummer, A. (2006). Model-in-the-loop testing. *Proceedings of The Institution of Mechanical Engineers Part I-journal of Systems and Control Engineering - PROC INST MECH ENG I-J SYST C*, 220, 183–199. doi: 10.1243/09596518JSCE207.

Souza, S. (2020). Micropython test lib. URL `https://github.com/saramonteiro/micropython_test_lib`.

Souza, S. and de Carvalho, R.A. (2021). Automated driver testing for small footprint embedded systems. *arXiv preprint arXiv:2105.01451*.

Zander, J. (2008). Model-based testing of real-time embedded systems in the automotive domain. *Fraunhofer FOKUS*.