UNIVERSITÀ DEGLI STUDI DI PALERMO

DIPARTIMENTO DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Development of the Test Access Port Driver of a Test During Burn-In Coverage Enhancement System for Automotive SoCs

TESI DI LAUREA DI

Morgan Lombardo

0673761

RELATORE

Prof. Daniele Peri

CORRELATORE

Giulio Zoppi

STMicroelectronics

CONTRORELATORE

Prof. Alessandra De Paola

ANNO ACCADEMICO 2019 – 2020

Development of the Test Access Port Driver of a Test During Burn-In Coverage

Tesi di Laurea Magistrale in Ingegneria Informatica

Morgan Lombardo

Relatore: Prof. Daniele Peri
Correlatore: Giulio Zoppi STMicroelectronics
Enhancement System for Automotive SoCs

Ringrazio l’aiuto datomi in questo periodo tantissime persone: primo fra tutti Giulio Zoppi, amico e mentore oltre che semplice tutor, per avermi supportato dall’inizio di questo percorso fino al momento in cui scrivo queste righe e a Daniele Peri, tutor universitario, per avermi aiutato.
Ringrazio chiunque in ST mi abbia aiutato a redigere questa tesi: Giorgio Pollaccia, per avermi fornito tutti i chiarimenti necessari ogni volta avessi qualche dubbio, per aver spronato e per avermi insegnato cosa veramente significhi dare il giusto peso al tempo, Francesco Camarda ed Emanuele di Miceli per avermi assistito ed aver provato constantemente il codice durante lo sviluppo del mio progetto.
Ringrazio Giuseppe Compagno, responsabile della sede di Palermo, per avermi accolto come fossi a casa sia durante l’esperienza di tirocinio che durante la stesura della tesi, insieme a tutti gli altri dipendenti della stessa sede, ogni parola, aiuto, concetto che mi avete detto o insegnato (al lavoro o al ristorante durante le pause pranzo) ha reso possibile tutto ciò.
Infine, ringrazio anche Giuseppe Di Giore, della ST di Catania, per aver fatto in modo che io avessi tutto il materiale tecnico per continuare a sviluppare la tesi.
Naturalmente non ci sono solo loro.
Ringrazio la mia famiglia, mia madre e mio padre principalemente, per avermi sempre motivato e per essermi stati accanto anche durante il momento più buio: “tranquillo, ogni impedimento è giovamento”, ora so veramente cosa vogliano dire queste parole.
Ringrazio Serena, la mia ragazza, per essere stata sempre pronta a darmi una parola di conforto e per avermi sempre ricordato che se sono arrivato a questo punto, allora potevo anche fare tutto questo.
Ringrazio Laura, sorella più che amica: “muoviti a laurearti, che devi diventare ricco” è stato il mio mantra, ed è più profondo di quel che sembra.
Ringrazio infine Martina per avermi dato costantemente l’aiuto necessario correggendomi dove necessario.
Ogni parola di questo lavoro è dedicata a tutti voi.

ABSTRACT

A microcontroller is a complete system that integrates in a chip a CPU, RAM and flash memory and some peripherals that are able, by means of I/O pins, to interface with the external world.

With the continuous development of new peripherals and new technologies, the microcontrollers have become widely used in any field, ranging from almost every electronic device up to those devices that were originally not designed to mount electronic components, for instance the cars.

With the constant increase of the usage of the microcontrollers some characteristics have been improved: the computing power, the available memory, the number of peripherals and the reliability of the same. The reliability is defined as the certainty that a microcontroller is defect free and it will not break during the usage[1]. To fit these requirements is necessary to develop some techniques and methodologies that allow the engineers to ensure that the chip is defect free, including the ones discussed in this thesis: the JTAG and the ATPG.

Joint Test Action Group (JTAG) and Automatic Test Pattern Generation (ATPG) are, respectively, an industrial standard the former and a technology the latter that are used together during devices testing phase. With JTAG is possible, with the use of a special circuitry composed by a set of registers and data lines, named Test Access Port (TAP), and a set of manufactured-defined instructions, to bring the tested device into a special working mode named *test mode* where is possible to read or write the RAM or the flash memory or to reconfigure the device (or do some other special operations not allowed on normal working mode, named *user mode*), meanwhile the purpose of ATPG is to mathematically model some patterns with the usage of an external software, to transmit them to the tested device, which before was reconfigured via JTAG, in order to trigger the defects. These tests are carried out with the ultimate goal of breaking the defective device, so as not to put it on the market, in addition ATPG along with technologies like Built-In Self-Test (BIST) and methods like BurnIn allows with minimal hardware (and therefore with reduced costs) to do the tests despite their increasing complexity and small size. All these

tests are done before doing a final test, studied to ensure the correct functionality of the device in its final working environment: the System Level Test (SLT).

This paper discusses the development of the JTAG and the ATPG modules for the SPC5 family of microcontrollers, carried out during my internship at STMicroelectronics in Palermo with the Automotive And Discrete Group (ADG), done as an activity to support their project of the test platform named Burnin+SLT.

The modules were developed following a process of constant improvement both in the software side, writing a code that meets the criteria of structure, portability and security, and in the hardware side considering which peripherals are best suited to the purpose and require a limited amount of resources.

The final purposes of this thesis are: first is to allow to a specific microcontroller, named supervisor, to communicate via JTAG standard with the Device Under Test (DUT) in order to bring it into test mode and configure it, the second is to allow to the microcontroller to manage the received ATPG patterns by transmitting it to the DUT or storing it into an external flash memory and the last purpose is to have something that can be easily reused on several microcontrollers or easily adapted to the working context.

Chapter one will clarify the reasons for the tests and will discuss about some used testing techniques, then the JTAG standard and ATPG technology will be introduced, followed finally by the chapters that show the choices made in hardware related to the communication protocols used, the tools used for the development of modules and the architecture of both modules. On last chapter, will be shown some transmissions done by the usage of the modules step by step.

Table of Contents

List of figures.......... 7
List of tables.......... 10
List of terms.......... 11
Overview of testing concepts.......... 13
1.0 State of the art.......... 13
1.1 Parameters and economy of testing of devices.......... 15
1.2 Testing methodologies.......... 16
The JTAG and ATPG.......... 25
2.0 Chapter introduction.......... 25
2.1 What is JTAG and advantages on using it.......... 25
2.2 JTAG interface.......... 27
2.3 Logical aspects of JTAG.......... 28
2.4 ATPG and Scan chain.......... 33
Communication interfaces.......... 38
3.1 Chapter introduction.......... 38
3.2 Used communication peripherals.......... 38
3.3 Serial Peripheral Interface (SPI).......... 39
3.4 Generic Timer Module.......... 43
3.5 Hardware configuration for the implemented modules.......... 47
Real Time Operating System (RTOS).......... 49
4.1 Chapter introduction.......... 49
4.2 Defining RTOS and advantages on using it... 49
4.3 RTOS concepts.......... 51
4.4 FreeRTOS.......... 54
4.5 Tasks communication ways.......... 55
4.6 The FreeRTOS CLI.......... 57
Programming software.......... 63
5.1 Chapter introduction.......... 63
5.2 Programming a microcontroller.......... 63
5.3 The configurator.......... 64
5.4 The IDE.......... 66
5.5 The debugger.......... 66

Software Implementation ... 68
6.1 Chapter introduction ... 68
6.2 Targets ... 69
6.3 Modules architecture and configuration ... 71
6.4 FreeRTOS integration ... 73
6.5 Low level driver functions ... 75
JTAG module ... 77
7.1 Chapter introduction ... 77
7.2 JTAG data type ... 77
7.3 JTAGops structure ... 80
7.4 JTAG module functions ... 81
7.5 How data to transmit is managed ... 83
7.6 JTAG task ... 90
ATPG module ... 93
8.1 Chapter introduction ... 93
8.2 ATPG data types ... 94
8.3 ATPGOPS structure ... 95
8.4 ATPG module functions ... 95
8.5 Data transmission ... 97
8.6 ATPG task ... 98
8.7 Sending data with GTM device ... 101
Guided User Interface (GUI) ... 110
9.1 Chapter introduction ... 110
9.2 GUI architecture ... 110
9.3 jFile class ... 112
9.4: jSerial class ... 114
9.5 The interface ... 116
Conclusions ... 119
10.1 Required resources ... 120
10.2 The usage of the GTM peripheral. ... 121
10.3 Modules operation ... 126
10.4 Improvements ... 133
References ... 134

List of figures

Figure 1.0: Crosstalk noise equivalent circuits…………………………………14
Figure 1.1: International Technology Roadmap for Semiconductors….……….15
Figure 1.2: Well-formed stray vs bad formed stray………………………..…..19
Figure 1.3: Bathtub curve…………………………..………………………..20
Figure 2.0: JTAG state machine……………………………......……………28
Figure 2.1: Tap output control connection………………………………………31
Figure 2.2: Types of faults detected by connection test…………………………34
Figure 2.3 Scan chain……………………………………………………......35
Figure 3.0: SPI connections master/slave……………………………………..40
Figure 3.1: Daisy chain and independent SPI configuration……………………42
Figure 3.2: Possible clock configuration………………………………………...43
Figure 4.0: Task states…………………………………………………………...53
Figure 4.1: Commands linked list example……………………………………...58
Figure 4.2: Command function example………………………………………...60
Figure 4.3: CLI flow……………………………………………………………..61
Figure 5.0 CubeIDE and SPC5Studio configurators……………………………65
Figure 6.0: SPC58EC-DISP board and STM32F446RE board.…………..……70
Figure 6.1: Modules architecture………………………………………………...72
Figure 6.2: Task priorities order…………………………………………………73
Figure 6.3: Data is send from TMS buffer and at same time received from TDO line……………………………………………………………...…………..76
Figure 7.0: JTAG low level driver structure……………………………………..79
Figure 7.1: JTAG driver structure………………………………………………..79
Figure 7.2: jtagops_t structure…………………………………………………...80
Figure 7.3: JTAG module function prototypes…………………………..…....83
Figure 7.4: JTAG_setTDI prototype……………………………………………..84
Figure 7.5: JTAG_setTDI function………………………………………………85
Figure 7.6: JTAG_setTMS function……………………………………………..87
Figure 7.7: TMS and TDI buffers before transmission…………………………89
Figure 7.8: Buffers before transmission on debug session………………………89

Figure 7.9: Transmitted TMS and TDI……………………………………………..89
Figure 7.10: TMS buffer filled to bring state machine to TEST_LOGIC/RESET state……………………………………………………………………………....90
Figure 7.11: JTAG task flow…………………………………………………….92
Figure 8.0: ATPG function prototypes……………………………………...97
Figure 8.1: Code executed for one shot transmission …………………………...99
Figure 8.2: Code executed for chunk transmissions……………………………..99
Figure 8.3: ATPG task flow chart …………………………………………...100
Figure 8.4: Data transmission example ($0xB0_{16}$)………………………………103
Figure 8.5: Bit CM0+1 detail…………..103
Figure 8.6: Data routing to ATOM module……………………………………104
Figure 8.7: MCS code flow……………………………………………………..108
Figure 8.11: MCS assembly code for channel 0 and channel 1………………...109
Figure 9.0: GUI architecture……………………………………………………111
Figure 9.1: jSerial class constructor…………………………………………….116
Figure 9.2: The GUI…………………………………………………………….116
Figure 9.3: GUI connected with the microcontroller…………………………...117
Figure 9.4: Error on file read……………………………………………………118
Figure 10.0: SPI clock signal and data signal synchronization…………………122
Figure 10.1: GTM clock and data signal synchronization……………………...122
Figure 10.2: Sampling error ($0xBF_{16}$)……………………………………………..122
Figure 10.3: Doubled data bits transmission example…………………………123
Figure 10.4: Correct sampling ($0xBF_{16}$)...123
Figure 10.5: Overflow bit set…………………………………………….......124
Figure 10.6: Working flow…………………………………………………...126
Figure 10.7: List of available commands………………………………………127
Figure 10.8: ATPG task waiting for the data to transmit………………………128
Figure 10.9: More than one data packet transmitted……….…………………..128
Figure 10.10: Last data packet truncated..…..……………………………...129
Figure 10.11: Response from microcontroller………..…………..…..………129
Figure 10.12: jtag_read command transmission and response…….………...130
Figure 10.13: read ID printed on terminal……………………………………..130

Figure 10.14: tDataStructure after the command has been received…………..131

Figure 10.15: TDI buffer………………………………………………………131

Figure 10.16: TMS buffer……………………………………………………...132

Figure 10.17: JTAG transmission……………………………………………...132

List of tables

Table 1.0: Advantages and disadvantages of SLT…………………………….....23

Table 4.0: Differences between bare-metal and RTOS………………………......51

Table 10.1: Amount of required resources……………………………………....120

Table 10.2: Required hardware peripherals……………………………………..120

List of terms

AAC: Adaptive Cruise Control

ACB: Aru Control Bits

ADAS: Advanced Drive Assistant System

AEB: Autonomous Emergency Breaking

AFD: AEI to FIFO

ARU: Advanced Routing Unit

ATE: Automatic Test Equipment

ATOM: Aru-Tom connected

ATPG: Auto Test Pattern Generator

BGA: Ball Grid Array

BIST: Built-In Self Test

CFGU: Configurable Clock Generation Unit

CLI: Command Line Interpreter

CLK: Clock

CMU: Clock Management Unit

CPHA: Clock Phase

CPOL: Clock Polarity

CPU: Central Processing Unit

CS: Chip Select

CTRG: Clear Trigger Register

DFT: Design For Testability

DIY: Do It Yourself

DMA: Direct Memory Access

DPMM: Defective Parts Per Million

DUT: Device Under Test

ECU: External Clock Unit

F2A: FIFO to ARU

FXU: Fixed Clock Unit

GTM: Generic Timer Module

GU: Guided User Interface

IC: Integrated Circuit

IEEE: Institute of Electrical and Electronic Engineers

ISR: Interrupt Service Routine

ITRS: International Technology Roadmap for Semiconductors

JTAG: Joint Test Action Group

LDW: Lane Departure Warning

MCS: Multi-Channel Sequencer

MISO: Master Input Slave Output

MOSI: Master Output Slave Input

MSB: Most Significant Bit

MTBF: Mean Time Between Failures

MTTF: Mean Time To Failure

MTTR: Mean Time To Repair

OTP: One Time Programming

PSM: Parameter Storage Module

RAM: Random Access Memory

RTOS: Real-Time Operating System

SCLK: Scan Clock

SE: Scan Enable

SFF: Scan Flip-Flops

SI: Scan In

SIA: Semiconductor Industry Association

SLT: System Level Test

SO: Scan Out

SoC: System on Chip

SOMC: Signal Output Mode Compare

SOMI: Signal Output Mode Immediate

SOMP: Signal Output Mode PWM

SOMS: Signal Output Mode Serial

SPI: Serial Peripheral Interface

STRG: Set Trigger Register

TAP: Test Access Port

TBCM: Tim Bits Compression Mode

TCB: Task Control Block

TCLK: Test Clock

TDI: Test Data Input

TDO: Test Data Output

TIM: Timer Input Module

TMS: Test Mode Select

TOM: Timer Output Module

TRST: Test Reset

TSR: Traffic Sign Recognition

UART: Universal Asynchronous Receiver Transmitter

CHAPTER 1

Overview of testing concepts

1.0 State of the art

Since the invention of microcontrollers (1975) to date, their demand and fields in which they can be used have increased along with their computing power and complexity. The microcontrollers were initially used for specific tasks, they were equipped with memory that could be written only once, this type of memory is called One Time Programming (OTP) memory, and the code to execute was loaded inside them during the production phase, lacking in this way in versatility. Over the years, however, this trend has been reversed and today is possible to find microcontrollers everywhere: consumer electronics, clothing, medical, military and transport.

Considering the transport sector, and in particular the cars, the use of microcontrollers has replaced the electrical systems thanks to their low price and increasing reliability, allowing the installation of new features ranging from the ignition of the engine to the improvement of its efficiency up to the latest systems able to control the driving.[2]

The usage of microcontrollers on the cars has led to the birth of a specific sector: the automotive sector, born with the purpose to study and develop new systems to mount inside a car to improve the driving experience.

All these systems form the ADAS system, which stands for Advanced Drive Assistant System and it can comprise many subsystems, for instance:

- Adaptive Cruise Control (ACC)
- Lane Departure Warning (LDW)
- Autonomous Emergency Braking (AEB)
- Traffic Sign Recognition (TSR)

The various microcontrollers that make up these systems must not only meet the criteria of computing power or available memory, they must meet high standards of reliability too, defined as the certainty that the device is free of defects which may affect its functionality. The concept of reliability is fundamental and therefore it is also essential to continuously test the devices. In this context, the aim of the industries is not to ensure zero defectiveness during the production phase (too much expensive to achieve) but to ensure zero defectiveness towards the customer, in order to minimize the number of defective parts placed on the market. With growing in complexity of the microcontrollers, according to the Moore Law (1968), and with the shrinking in size of every new IC, ensure the chip integrity became much difficult, mainly because there are more problems that previously could be ignored. For instance, some of these problems fall into the category of *crosstalk noise* problems that can be divided into *signal integrity* and *delay degradation*: the former is an electrical disturbance caused by the reduced distance between two traces of a circuit, when a signal crosses a trace, it creates a crosstalk current in the nearby trace that can be detected by the devices connected to it, while the latter happens when the noise couples to a switching signal[3].
These and others are the problems that can affect the specific components within the device and if one of these will be triggered during drive, it can lead only to catastrophic results.

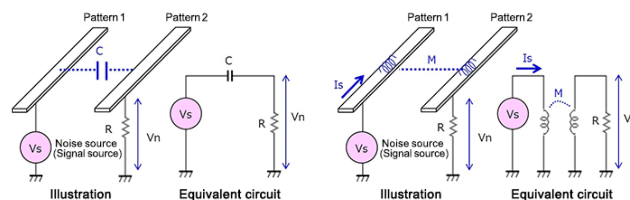

Figure 2.0 Crosstalk noise equivalent circuits [4]

1.1 Parameters and economy of testing of devices

To control the quality of the production, some parameters are considered:

- Yield rate: ratio of pieces that can be considered acceptable, calculated by the equation[5]

$$Yield\ Rate = \frac{n_accepted_pieces}{n_total_pieces}$$

- Reject rate: ratio of pieces that must be refused because they do not pass final test, calculated by the equation[5]:

$$Reject\ Rate = \frac{n_pieces_that_fails_final_test}{n_tot_tested_pieces}$$

In 2004, the Semiconductor Industry Association (SIA) published an International Technology Roadmap for Semiconductors (ITRS) that includes an update to the test and test equipment trends for nanometer designs through the year 2010 and beyond, showing that, at some point, the cost of the tests will surpass the cost of production.[5]

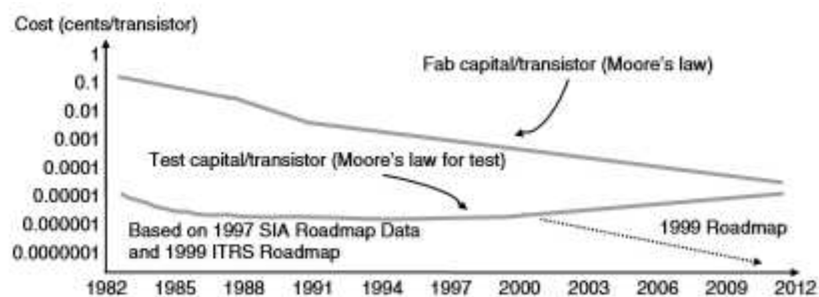

Figure 1.1 International Technology Roadmap for Semiconductors. [5]

To let devices selling price not grow too much, the manufacturers divide them into different groups depending on the final usage and, depending by the class, they are more or less tested. For instance, consider the class of microcontrollers customer-oriented and the class of microcontrollers used on automotive field: the

former, since they are addressed to the standard consumer and mainly used for less delicate jobs, are subject to lower reliability criteria so on the one hand, on the market are put pieces that can be defective and that can break shortly after the purchase, on the other hand, however, their production cost is low and consequently also the selling price. Although the class of microcontrollers used on automotive field, it is unthinkable that a piece is defective, therefore they must be tested much more tightly so them price will grow.
For instance, consider two different microcontrollers sold by ST: first one is the STM32F446RE, a customer oriented microcontroller sold at the price of 3.6$[6], and the other is the SPC58EC80E5 an automotive microcontroller, sold at the price of $14.1[7]. The first one is a consumer device, so its price is not high and it is more prone to faults than the second one which is a microcontroller used on automotive field so the seller must be sure that is not a faulty piece.

1.2 Testing methodologies

Defects in the production of the ICs can occur at any time and finding a defect on advanced production phase cost more than finding it in the same production phase in which it was generated.
There are a lot of different testing techniques, used on different production phases. Some of these are:

- IDDQ test: This test is typically done on the chips that uses a CMOS structure. The purpose of this test is to detect some defects like power shorting by detecting the presence or absence of a leak current named ICCLEAK. This test establish a current path from the current supply to the ground to detect internal circuit defects.
- AT-SPEED test: this test consists of a series of two-shot high-speed clock named *launch* and *capture.* They are used with purposes of testing timing failures and to measure the maximum working frequency of the IC.
- Cell Aware Test: is a transistor level test that is done to detect defects like short circuits and open circuits.[8]

1.3.1 DFT

Design For Testability (DFT) is a set of testing techniques used to test the devices based on some key factors:

- Observability: is the possibility to observe the output of a logical net (a logical net is considered as a set of logic gates that composes the device circuitry).
- Controllability: is the possibility to force to logic value 0 or 1 one pin of the device.

Does not exist a unique DFT technique that can be used for all the devices, so the existing different techniques are divided into two groups:

- Ad-hoc: these tests are developed for a specified device, often these tests are based on the engineers experience.[9]
- Structured: these tests follow a well-defined procedure and can be applied on different devices.

Structured testing techniques are the most used ones, they transform testing from a sequential one to combinational one, so this tests are easier to do.
Some types of structured test are:

- Scan.
- Partial-scan.
- Built In Se lf-Test (BIST).
- BurnIn.

In scan and partial scan, apposite circuitry is added to the IC to let the test be done, so one or more testing pins are used and the flip-flops, that compose the logic network, are replaced by scan flip-flops (SFF) that are connected in order to behave as a unique big shift register.
Input of the first flip-flop is directly connected with the input pin, the output of last flip-flop is directly connected with the output pin. Auto Pattern Test Generator (ATPG) is used to obtain tests for all testable faults in the combinational logic.

Another DFT technique is the Built-In Self-Test (BIST), that is a set of apposite circuitry that allows to an IC to test itself. The BIST is used to make faster and less expensive integrated circuit manufacturing tests. The IC has a function that verify all or a portion of its internal functionality, in the automotive field this type of test allows a specific component (for example ABS system) to test itself and communicate to the driver if it is working properly.

1.3.2 BurnIn

BurnIn test is done at last production phase. This test is based on two considerations about the defects and the lifetime of a device: intrinsic and extrinsic defects and the bathtub curve.

1.3.2.1 Defects classification

A defect can be classified as:

- Intrinsic: it is defined as a defect caused by a design error, such as a badly sized straw.
- Extrinsic: it is defined as a defect caused, for instance, by a malfunction of one of the machines that print the device. This kind of error is absolutely not predictable and cannot be reported.

Consider this situation:
A layouter that is working on a chip knows very well that a 90° degree corner trace can tolerate a limited amount of current in order to not be damaged (e.g 100 mA).
In case of no defects, the trace and the corner will be built with correct width, and the current amount will not exceed maximum limit.
An intrinsic error can appear if trace width is not well calculated. This error can be spotted and corrected during the layout evaluation and the size can be modified to the proper value prior to release the masks. So intrinsic errors are related to the design phase.

An extrinsic error is not related to the design of the IC, but is related to some anomalies during the production phase of the latter that could be not predicted or signaled.

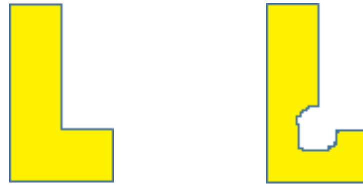

Figure 1.2 Well-formed trace vs bad formed trace

First trace is flawless, second one present one extrinsic defect. This defect reduces the width of stray on the corner, so the maximum amount of current will be no more 100 mA but less. This error can't be discovered because it is not controlled or its existence is unknown. BurnIn test on one hand will destroy this piece so it will not be put in the market, on other hand it will let the engineers to know the existence of this defect.

1.3.2.2 Bathtub curve

The second consideration for the BurnIn test is based on the bathtub curve that is constructed using a parameter named *failure rate*.

The *failure rate* depends by the considered system: it can be considered as Mean Time Between Failures (MTBF) or Mean Time To Repair (MTTR) if the system is repairable, Mean Time To Failure (MTTF) if the system cannot be repaired.

The *failure rate* changes with the course of time and the way it changes allow to build the bathtub curve.[10]

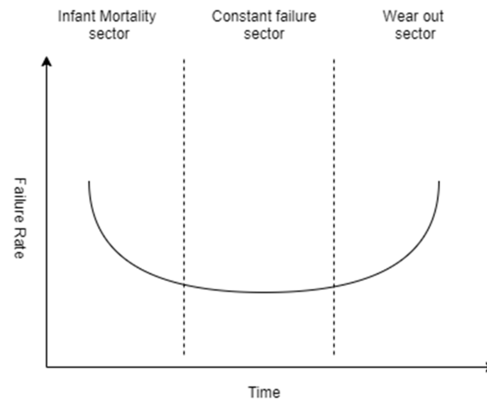

Figure 1.3 Bathtub curve

The bathtub curve represents the life span of an IC, and it is divided into three sectors: first one represents the first period, known as *infant mortality* sector, where is more plausible for an IC to fail if it have some defects, second period is the *constant failure* period, in this sector of the curve the IC should work properly and does not present any defect, if some defect is triggered during this phase is considered as random fault. The last sector is the *wear out*, here is more plausible for an IC to fail because of the wear out.

BurnIn is done in order to let chip pass the first period of life, the infant mortality period, in this way if a chip pass this period without broke itself then is plausible that it does not present any defect.

Another aspect to consider in order to test correctly the devices with BurnIn test is the mission profile, a set of electrical and thermal specifications defined by device manufacturer that are used to correctly set the parameters of BurnIn test: voltage, temperature and time.

1.3.2.3 Technical aspects of BurnIn

BunIn test is done adapting it to the testing specifications for every different tested device depending on mission profile.

BurnIn is done with a temperature of about 125 Celsius degrees, with a voltage that is 20% more than IC working voltage, but it can be 30% more if specified on mission profile, for some hours.

BurnIn can be made in two variants: from a voltage or from a temperature point of view. This aspect is important because some defects depend only on temperature, others depend on working voltage.
To age the DUT, BurnIn duration is calculated on the basis of two parameters: they are called temperature accelerator factor and voltage accelerator factor.
Temperature accelerator factor is calculated using Arrhenius equation:

$$AF_T = e^{[\left(\frac{E_{aa}}{k}\right)*\left(\frac{1}{T_U}-\frac{1}{T_A}\right)]} [11]$$

Where:
AF_T = Temperature acceleration factor
E_{aa} = Apparent activation energy in electron volts (eV)
k = Boltzmann's constant (8.617 × 10-5 electron volts/°Kelvin)
T_U = Junction temperature at normal use conditions in degrees Kelvin
T_A = Junction temperature at accelerated conditions in degrees Kelvin

Voltage accelerator factor is calculated using following equation:

$$AF_V = e^{[\left(\frac{K}{X}\right)*\left(\frac{V_A}{V_U}\right)]} [11]$$

Where:
AF_V = voltage acceleration factor
K = experimentally determined electric field constant (expressed in thickness per volt)
X = Thickness of stressed dielectric
V_A = Stress voltage in accelerated ELF test
V_U = Use voltage

So, these acceleration factors allow tester to know the actual use condition period named t_U, knowing test period indicated with t_A, as:

$$t_U = AF_T * t_A[11]$$

So, for instance, if the acceleration factor is AF_T = 1000 and we have done BurnIn on a DUT for 3 hours, with previous equation we can ensure that the DUT will be aged about 3000 hours. If the mission profile of the DUT says that the *infant mortality* sector is a period with a duration less than 3000 hours, and if the DUT pass this period without fail, the engineers can ensure that the tested device does not have any defect.
Last annotation is that for this test is important only to pass infant mortality period, so BurnIn test is never done in order to reach the *wear out* period of the DUT.

1.3.3 System Level Test

The device must be tested at every stage of production to ensure that any defect that occurs is immediately detected. The tests range from those that are carried out during the printing of the device circuitry on the wafer, such as iddq-test, to the BurnIn, which is carried out after the IC has been mounted in the package, all these tests are done to reduce the Defective Parts Per Million (DPMM) ratio.
During the tests, there was a real difference between the performance of the latter during the testing phase and the operational phase.[12]
To reduce this difference the companies are slowly introducing into their test chain a last test to do when the production phases are over: the System Level Test (SLT).
SLT consists in verifying the functionality of the device by placing it in an environment that simulates as much as possible the final working environment so as to make it interact with various devices or peripherals and stress it by applying functional patterns.[13]

The advantages and disadvantages of the SLT are:

Advantages	Disadvantages
Is based on a relatively cheap test environment.[14]	Time is a critical parameter for this test: it spans in the range of minutes.[14]
Is a flexible test: is easy to modify the test by adding new parts.[14]	Even if the cost is reduced, it is still a factor that negatively affects the entry of the SLT in the testing chain, in addition, the increase of testing time let the cost grows too.

Table 1.0: Advantages and disadvantages of SLT.

These aspects slow down the introduction of SLT in the testing chain, even if new solutions are being tried, for instance the union of the SLT to BurnIn[12] thus managing to cut costs by exploiting their similarities (for instance the fact that both tests take time).
This solution is being designed by the ST.
The union of two tests into a single one includes the modification of the board where the devices that must be subjected to BurnIn are placed in order to contain also the necessary logic to execute the SLT. The studied architecture aims to offer mainly two advantages in order to reduce the times and costs of the test: the former is an high parallelism, which allows to put on the same BuirnIn board more DUTs together and test them at the same time, the latter is the possibility to transmit JTAG signals to all the DUTs at the same time, provided that enough bandwidth can be ensured in order to transmit and receive data to and from all the devices together.[15]
The BurnIn+SLT architecture designed (still in development) consists of two entities: the former is the DUT mounted near all the needed circuitry used to simulate the functional environment in order to do the SLT, the latter is the supervisor microcontroller, together with all the circuitry necessary for collection of data and measurements concerning the conduct of the test and for the

communication with between the supervisor and the DUT or between the supervisor and the terminal.[12]

CHAPTER 2

The JTAG and ATPG

2.0 Chapter introduction

Traditional testing techniques involve testing a device by “touching” the various sectors of the combinational net that compose it to verify its operation. As the size of the tested device decreases, along with the increase in complexity and therefore in the number of used transistors and the I/O pins that make it up, the physical approach becomes increasingly difficult and it is therefore necessary to look for an approach for testing that does not act from outside, but that is an integral part of the device. To resolve this problem, a consortium of about 200 industries has defined the JTAG standard.

Together with the needed circuitry for the JTAG standard, which is discussed on following paragraphs, with the integration within the chips of an additional small amount of circuitry, the engineers can exploit the ATPG, whose purpose is to test the logical networks of a sector of a chip by stimulating them and checking if some defect has been triggered. This chapter will first introduce and discuss the JTAG standard, followed by the ATPG methodology.

2.1 What is JTAG and advantages on using it

Joint Test Action Group (JTAG) is an industrial standard defined by the Institute of Electrical and Electronic Engineers (IEEE) 1149.1, created between 1985 and 1990 with the first revisions released respectively in 1993 and 1994, which allows the IC testing by implementing directly in hardware the necessary circuitry.

With JTAG is possible to set the device on a special working mode called *test mode* that is different from the traditional working mode, called *user mode*.

When the device works in *test mode* is possible to read or write new values inside the configuration registers, write or read the flash and RAM memories during the normal code execution (on *user mode*, code execution must be stopped before

changing the configuration of the microcontroller) or access to some special debug functions disabled on *user mode*.

Because the JTAG allow to the device to enter into debug mode and so letting the user to have total control over it, entering in this mode is protected by a set of special instructions and data that must be transmitted to the microcontroller together with other precise steps to be carried out in precise time, which normally are maintained secret to the customer.

The advantages of using the JTAG are:

- Traditional testing: traditional board-level and device-level testing consumes a greater amount of time, and in addition, for every device could be necessary to include apposite testing machines, increasing in this way the cost of tests. Extensive testing is mandatory for fields like defence, aerospace and automotive and here JTAG is beneficial in terms of testing, time and cost.
- Efficient testing: incorporating design-for-test techniques allows to do embedded testing, which give the possibility to read data scanned out while other data is scanned in, in order to stimulate chip internal nodes, saving in this way time.
- Lower costs: The additional cost of designing testability into a system during the design phase contributes to increasing the product lifespan, guaranteeing the possibility to test it, update it or detect failures even after it has been sold. Furthermore, JTAG allows to use a standard test approach that not change changing the SoC that is under test, so is not necessary to use different circuitry for different devices, again reducing cost.
- Board-Level isolation: if the device to test have a complex circuitry, with a large number of IPs, the JTAG together with the use of “boundary scannable” devices let possible to divide the IC into partitions, letting fault isolation easier to do.
- Simple Access: new chips that are built with new methods and technologies are very difficult to test “touching” them or accessing them using manual probes or ATEs. Testability with boundary scan architecture eliminates the problems related to physical access because all the necessary tools for test

are already mounted on the PCB near the chip, so the only connection needed to test chip is the 4-wire connector that will be used to send and receive data.

2.2 JTAG interface

The JTAG interface has a number of lines that are used together with some special circuitry, composed mainly by a set of multiplexers and registers, this set of electrical components is named Test Access Port (TAP).

The JTAG port is used for JTAG control as well as providing connections by which the serial data may enter and leave the board.

To communicate with the DUT, 4 signals are mandatory and 1 is optional:

- TCK (Test clock): clock signal. It times both the state transaction of the TAP and the input/output data shifting.
- TMS (Test mode select): value on this line selects the state of the TAP.
- TDI (Test data in): serial input data for instruction register and data registers.
- TDO (Test data out): serial output data for instruction and data registers, every bit is shifted out at the falling edge of the TCK.
- TRST (Test Reset): this line is optional, it is used to drive to reset state the TAP, but is possible to drive it on reset state by driving TMS line to 1 for five continuous clock cycles[16] .

2.3 Logical aspects of JTAG

The TMS pin is used to drive the JTAG state machine across its states. This allows user to change operation mode and read or write instruction or data from the TAP registers.

The JTAG state machine is defined by 16 states, divided into 3 groups: first one is a set of idle states, where all registers are resected and TAP controller execute the previously loaded instructions or tests, second one is a set of states that control data flow over TAP controller, third one is a set of states that control the instruction flow over TAP controller.

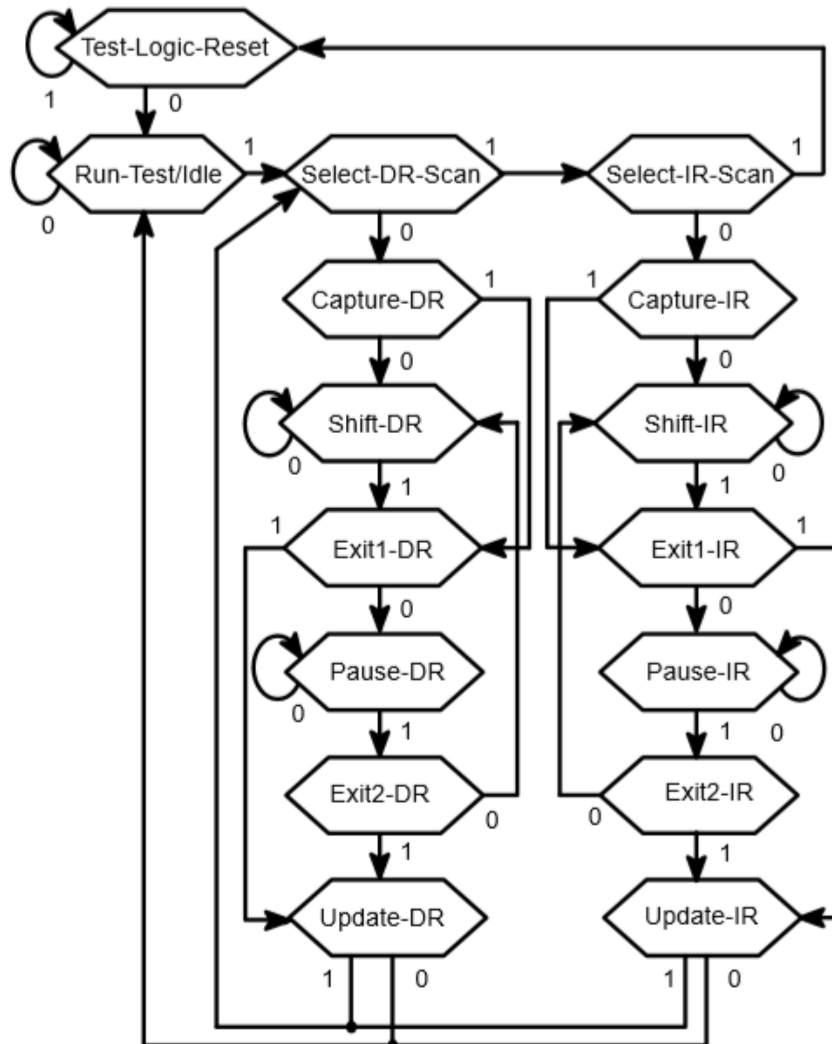

Figure 2.0: JTAG state machine [17]

First Group:

- TEST-LOGIC-RESET: when the state machine reaches this state, then the instruction registers are resected to their default value. When in this state, IC run normally.
- RUN-TEST/IDLE: when state machine reaches this state, it starts the debug, so the TAP is ready to shift data or instructions.[17]

Second Group:

- SELECT DR/SCAN: this state is entered prior to perform a scan operation on data register.
- CAPTURE DR: this state allows selected data register to shift data in/out on rising edge of TCK.
- SHIFT DR: when TAP controller enter on this state, it shifts data bits on data register.
- EXIT1 DR: when this state is reached, data shift is terminated.
- PAUSE DR: when this state is reached, shift on data register is paused until next state is reached.
- EXIT2 DR: same as Exit1, but it allows to update data register.
- UPDATE DR: when this state is reached, the shifted data will be latched on parallel output.[17]

Third Group:

- SELECT IR/SCAN: this state is entered prior perform a scan operation to instruction register.
- CAPTURE IR: this state allows parallel input to be loaded into instruction register.
- SHIFT IR: this state allows instruction bit to be loaded on instruction register.
- EXIT1 IR: when this state is reached, instruction shift is terminated.

- PAUSE IR: when this state is reached, the shift on instruction register is paused until next state is reached.
- EXIT2 IR: same as Exit1, but it allows to update instruction register.
- UPDATE IR: when this state is reached, the shifted instruction will be latched on parallel output.[17]

Test-Logic-Reset can be reached from any state by shifting five 1s (0x1F) on the TMS line, or can be reached using TRST pin if present.

The states of the second group, data register, and third group, instruction register, are symmetric. For the data registers, capture operation is started by entering on CAPTURE-DR state, then the data is loaded into the selected serial data path.

In the instruction register, the CAPTURE-IR state is used to capture status information into the instruction register. From the capture state, the TAP transition to either the SHIFT or EXIT1 state, normally the shift state follows the capture state so that test data or status information can be shifted into the appropriate register. Following the shift state, the TAP can return to RUN-TEST/IDLE via the EXIT1 state and the UPDATE state or enter to the PAUSE state via EXIT1 state. EXIT2 state is used to return from the PAUSE state to the CAPTURE-DR/IR state, if new data to send was uploaded and must be shifted inside the data register, or it can be used to go to UPDATE-DR/IR state. Upon entering into a data or instruction scan block the hidden latches are forced to hold last loaded value, in this way to load new data inside the shadow latches is necessary to drive the state machine through UPDATE-IR/UPDATE-DR state.

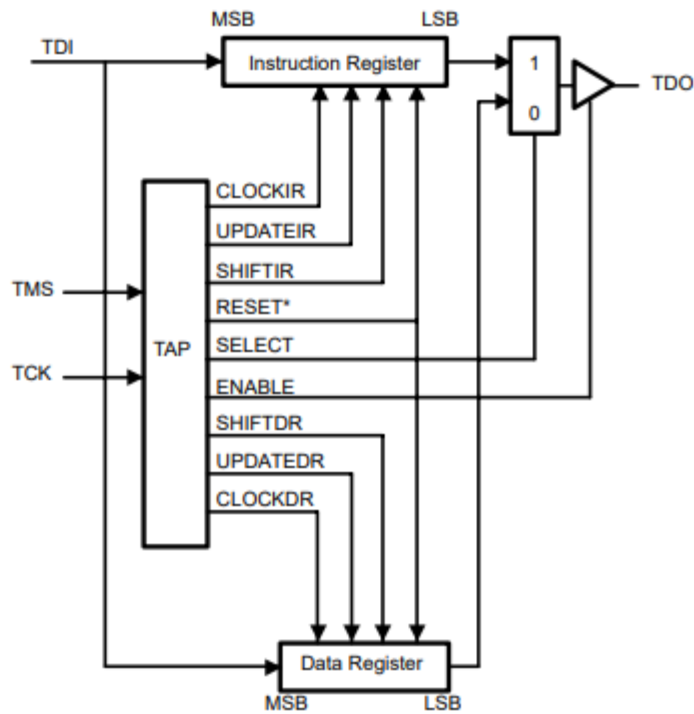

Figure 2.1: Tap output control connection.[17]

How image shows, almost 2 registers are needed to ensure correct work of TAP controller: Instruction register and Data register. More registers can optionally be added, depending on test purpose or chip architecture.

- Instruction register: Instruction register provides the address used to refer to one of the data registers present on the device architecture. So, the opcode of an instruction is just an address that drive a component, often a multiplexer, allowing data to be loaded into the data register selected from the instruction opcode.

  Instruction register is composed by two registers: one instruction register and one shadow instruction register, where data is loaded during the UPDATE-IR state. During instruction register scan, the instruction code is loaded into the instruction register via the instruction register shift enable SHIFT-IR and instruction register clock. During the rising edge of TCK, instruction code is preloaded and shifted in, meanwhile during falling edge, data is shifted out. Status inputs are user-defined, but at least 2 bits must be set, because an instruction has a minimal length of 2 bits. The instruction shadow register is composed by latches, one for every bit. On SHIFT-IR state, latches maintain

their state, that is updated only when the state machine reaches UPDATE-IR state, if RESET is activated, latches are loaded whit their BYPASS instruction code.[17]

- Data register: IEEE Std 1149.1 requires 2 data registers: the former is the boundary-scan register, the latter is the bypass register, a third optional register, IDCODE register, can be added too if TAP support IDCODE instruction. If necessary, more used-defined data registers can be added. Data registers are arranged in parallel from the TDI input and TDO output. During a data register scan operation (SCAN-DR state), the addressed scan register (the instruction opcode is the address of a data register) receives TAP control via the shift data state (SHIFT-DR) and the addressed register is filled whit data bits. During data register scan operation, SELECT-DR output from the TAP selects the output of the data register to drive TDO pin.[17]

Every device has his own JTAG instruction set, so to use the JTAG on a specific chip, is mandatory to know which instructions are implemented. Only some instructions are mandatory, they are:[16]

- BYPASS: when this instruction is received, the TDI line and TDO line are connected by the bypass register. During the bypass instruction, the IC works normally. This instruction is defined by a series of bits to one.
- SAMPLE/PRELOAD: this instruction allows TDI line and TDO line to be connected with one of the data registers. In this way is possible to take data that is entering and leaving the IC.
- EXTEST: Places the IC into external boundary scan mode and select the register to be connected with TDI line and TDO line. The opcode of this instruction is defined by a series of zeros.

2.4 ATPG and Scan chain

During device test phase, together with JTAG, the ATPG is used in order to test the logical networks that compose the circuitry.

Auto Test Pattern Generator (ATPG) is a technology that, by mathematically modeling the possible defects, creates some bit patterns that are transmitted to the tested device. The patterns shall be constructed in such a way that the response of a functioning device can be distinguished from the response of a defective device. These patterns are transmitted to the DUT using specific lines, named Scan In (SI) for input data, Scan Out (SO) for output data, Scan Enable (SE) to enable testing and Scan Clock (SCK) to give the clock signal to the components. These lines are connected to a set of scan flip-flops, mounted between two logical nets, creating in this way a unique big shift register named Scan Chain.

The ATPG effectiveness is measured according to the number of defects modelled and the coverage of scan network, in fact, for large circuitry, a modular approach is adopted so the scan networks are built in such a way as to cover a specific area while other areas are left uncovered, and, for architectural reasons, they cannot be tested.

With the combined use of ATPG and JTAG, is possible to do two different tests: connection test and scan chain test.

2.4.1 Connection Test

A connection test checks the interconnections between the components. This test can find defects like short circuits or open circuits. Some fault examples are shown below.

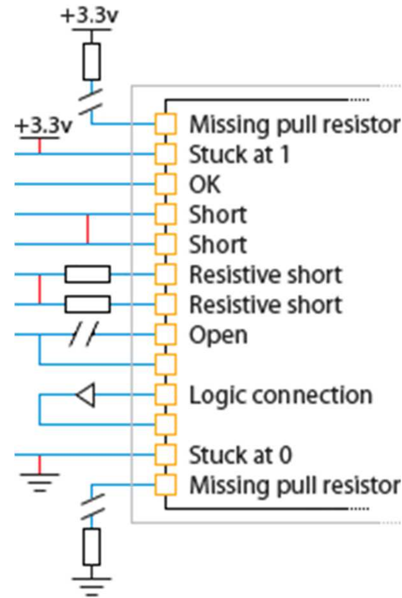

Figure 2.2: Types of faults detected by connection test.[18]

The main advantage of this test is its application in devices that are built with Ball Grid Array (BGA) technology, but, on other hand, if this test is carried out with the JTAG, it can affect only those components, inside the device, that mount JTAG circuitry.

2.4.2 Scan Chain test

The scan chain test is done in order to test interconnects within a sequential circuit, named network, using the patterns generated by ATPG software. If the network does not have defects, then the returned pattern will be the same of the expected one.

A scan chain is composed by a set of flip flops named scan flip-flop (SFF), connected with a piece of sequential logic and connected with another scan flip-

flop in order to create a big shift register. During this test, the ATPG pattern is shifted inside the IC along the scan flip flops, every bit that composes the pattern will stimulate the net between two scan flip-flops and, depending on the test phase, the value returned by the network after the stimuli can be stored inside the scan flip-flop mounted over the network.

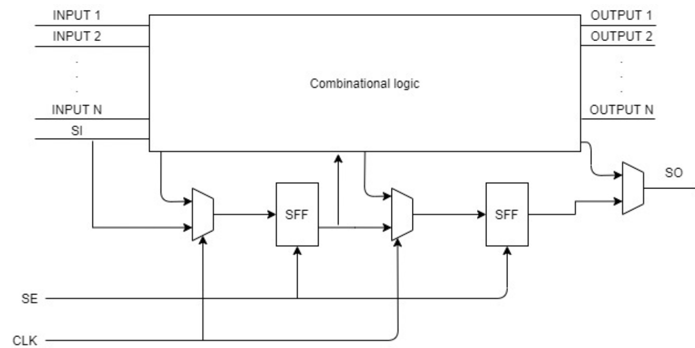

Figure 2.3: Scan chain

The test is divided into two phases: shift phase and shift+capture phase.

- Shift phase: During this phase, the stimulus are shifted inside the scan chain flip flops, every bit shifted inside will go into the net too, so the net state will evolve and it will be stressed in order to trigger defects. Net output will not be stored inside the flip flops during this phase so this test phase ensures the controllability of the network but not the observability (data is shifted in so the state of every net can be controlled, but the output is not stored so it is not observable).
- Shift+capture phase: During this phase, the output of the network is stored inside the flip flops and it is shifted out. If the output matches the expected one then the test is passed, if not a defect is discovered. This test phase ensures both the controllability and the observability of the net.

2.4.3 What is an ATPG pattern?

ATPG patterns are long sequences of bits mathematically modelled by a special software and transmitted to the tested device in order to stimulate the defects.
As already mentioned, the scan chain is a sequence of flip flops that form a single shift register, the patterns are shifted inside and then shifted out and, if pattern read in output is different from the expected one, then is possible to ensure that a defect has been triggered.
Today two types of ATPG test exist: first one is the combinational ATPG, where the nodes (a node is a flip-flop of the circuit) of a circuit are tested individually, allowing use of a simple vector matrix in order to quickly test all the comprising flip-flop.
Second ATPG test is the sequential ATPG that searches for a sequence of test vectors to detect a particular fault through the space of all possible test vector sequences, using some search strategies or heuristics to find a shorter sequence.[19]
Patterns are created with the use of mathematical models that represent the possible defects present in a device, some of them are:

- Stuck-at faults: when the node is blocked to the logical value 0 or 1.
- Slow faults: caused by setup-time violations.
- Fast faults: caused by the hold-time violations.[20]

The patterns are generated by algorithms that with the use of heuristics and a space of states, whose size varies according to the used algorithm, search in the latter the pattern that best simulates a defect. The most used algorithms for the generation of the patterns are three: the D Algorithm, the PODEM and the FAN.

- D Algorithm: the first implemented algorithm used to generate the patterns. In order to generate the patterns, this algorithm uses the information about the primitive nodes and the nodes that compose the input or output nodes of a subnet, named term or cubes, which make up the logical network to be tested.
- PODEM: stands for *path-oriented decision making*, it differs mainly from the D Algorithm because while the latter considers each node that makes up

the entire network to be tested, the PODEM algorithm considers only the primary nodes, for instance the nodes that input the network and the various sub networks that make up the network to be tested (each sub network can be broken down into a series of primary nodes).

- FAN: an improvement of the PODEM algorithm. Reduces the number of backtracks performed, optimizing the search time.[21]

Chapter 3

Communication interfaces

3.1 Chapter introduction

An important aspect of the development process of the JTAG and ATPG modules is the choice of the peripheral used to transmit data between the supervisor microcontroller and the DUT. The peripheral used must meet the requirements in accordance with the JTAG and the ATPG hardware architecture. In the following paragraphs will be discussed the criteria and the chosen peripheral, will also be discussed the architecture of the Generic Timer Module (GTM) device too, an advanced device designed by Bosch and used on automotive field as coprocessor, while in the following chapters will be evaluated the possibility of its use to transmit ATPG patterns.

3.2 Used communication peripherals

To find the most suitable communication peripheral, the following factors were considered:

- DUT that receives data, stores it into a shift register, so used peripheral should transmit data using a shift register too.
- There is no upper bound about transmission frequency.
- To let the test be faster, the used transmission peripheral should receive a bit in input for each output bit.
- There is no addressing.
- Depending on the instruction length or scan chain length, may need to send a number of bits that is not multiple of eight.
- If the JTAG is used, then two signals (TDI signal and TMS signal) must be shifted out together at the same clock frequency.
- Simplicity of use is a key factor too.

Considering these aspects, the communication peripheral that better fit the requirements is the Serial Peripheral Interface (SPI) both for the JTAG and ATPG transmission types. Another solution could be the use of the GTM device to transmit the ATPG patterns but its usage will be discussed in the last chapter.
The transmission with the SPI peripheral can be efficiently managed by using the interrupts system or the Direct Memory Access (DMA) circuitry.
The GTM, instead, is a special device composed by a set of submodules capable to do some complex operations without any CPU interaction. The purpose of the use of the GTM device is to automate the transmission process of the ATPG patterns, which can be very long.

3.3 Serial Peripheral Interface (SPI)

The Serial Peripheral interface (SPI) is a synchronous serial communication system widely used to ensure communication between a microcontroller and other devices or peripherals like memories or sensors. The SPI devices communicate using full-duplex transmission, thus transmission and reception occur simultaneously, and this happens between two entities: a master device and a slave device.
The master device uses a line named chip select (CS) to select the slave device that will send or receive the data, when data is transmitted it is stored inside a shift register, so for every bit that enters into the shift register, one bit goes out.
SPI communication is driven by 4 signals:

- CLK: SPI is a synchronous communication system, so a clock signal is mandatory to synchronize the master device and the slave device.
- MOSI: Master Output Slave Input is the line where data bits leave the master shift register and enter into slave shift register.
- MISO: Master Input Slave Output is the line where data bits go from the slave device to the master device.
- CS: Chip Select line is used in case more than one slave device is connected to master, this line allows to the master to select what slave device will receive or transmit data. Depending on connection between the master and

the slaves, one CS line for every slave device must be used. This line is named slave select (SS) too.[22]

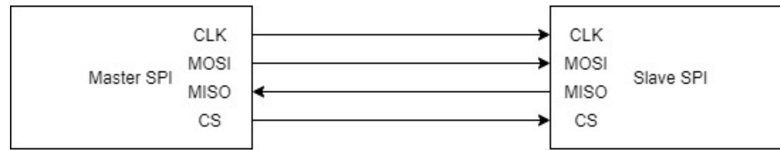

Figure 3.0: SPI connections master/slave

To begin the communication, first thing first the master device must select a clock frequency depending on the slave device, communication speed can be up to a few Mhz.

During each SPI clock cycle, a full-duplex data transmission occurs: when the master sends a bit on MOSI line, the slave receives it and sends a bit out on MISO line.

The transmission involves two shift registers of the same size and on most microcontrollers is possible to select the size of the data frame to transmit and typically it is of eight bits. Clock edge configuration defines when the data is shifted and sampled. When the transmission ends, the master device will deselect the slave device.

Data transmission can be done without CPU interaction, enabling the use of Direct Memory Access (DMA).

DMA is a special circuitry that connects one peripheral data register with the memory in order to automate the moving process, by moving one block of data at a time, from the data register to the memory and vice versa. The DMA is composed by a DMA controller (DMAC) that manages the data flow, a set of registers that can be used to set some transmission parameters, like the data source and the data destination, the number of bytes to move and the operation to do (read operation or write operation).

The DMAC checks if data can be moved from the source to the destination by checking CPU operation. If the current instruction does not involve the use of the data bus, then the DMAC starts the data transfer from the source to the destination.

This is done until the number of moved blocks is equal to the number written in the block number register.
The DMA uses only few interrupts to signal to the CPU the transfer status, normally the interrupts are used to signal: transfer started, half transfer complete, transfer complete or errors.

3.3.1 Connections between master and slaves

The SPI offers two ways to connect the master device and the slave devices: independent slave configuration and daisy-chain slave configuration. The choice of the configuration to use should be done basing it on the number of free pins to use as CS line.

- Independent slave configuration: in this configuration, the slave device that will communicate with the master is selected by the latter with a separated CS line for every slave device.
  The slave devices share the same MOSI line and is mandatory for every slave that its MOSI pin is tri-state to prevent data go on unselected slave. Pull-up resistors are recommended too between the CS line and the power line to prevent it to be on undefined state. The advantage of this configuration is the communication speed, on the other hand, for every slave device connected to the master is necessary to use a pin of the latter in order to control the slaves CS line.
- Daisy-chain: when the slave devices are configured in daisy-chain, the MISO line of the first slave device is connected with the MOSI line of the second slave device, and so on for the other slaves.
  In this configuration every device acts as unique shift register. The advantage of this configuration is that only one CS line is necessary for all the slaves, on the other hand, the amount of data bits to send grows accordingly with the shift register size of every slave device connected to the chain. Data bits to be sent can be calculated with the formula: $i * frame_size$, where i is the index of the slave device that should receive data (i.e for the first slave index is 1, for second slave index is 2 and so on) and $frame_size$ is the data size.

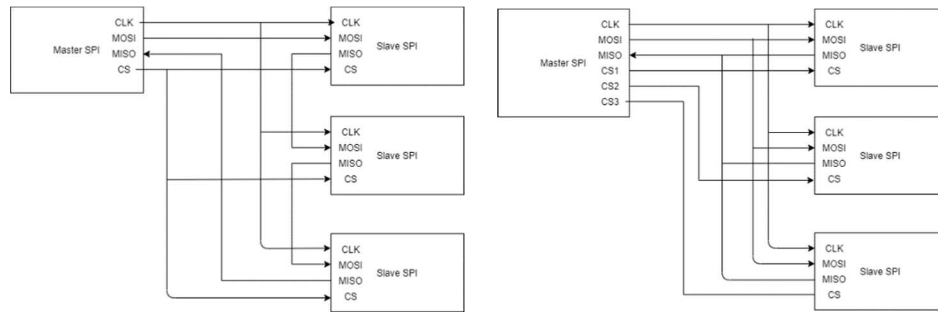

Figure 3.1: Daisy chain and independent SPI configuration

3.3.2 SPI clock configurations

For every clock cycle, one operation occurs on both clock edges: on the first edge data is sampled, on the second edge the sampled data is shifted into the shift register. Default SPI configuration samples the data on the rising edge of the clock and shifts it during the falling edge of the clock.

The SPI protocol allows to change this configuration by setting two parameters named respectively CPOL and CPHA.

- CPOL: it manages the clock polarity. If CPOL = 0, is a clock with idle state to 0, so every clock pulse is a 1.

  If CPOL = 1, then the clock idle state is to 1, so every clock pulse is a 0.
- CPHA: it determines the timing of master and slave devices operation. When CPHA = 0, a data bit is sampled during the rising clock edge and it is shifted during the falling clock edge. On the master side, data is changed during the trailing edge of the clock cycle.

  The combination of clock polarity and clock phase are often referred to as modes which are commonly numbered according to the combination of CPOL and CPHA, they go from 0 to 3 (00_2, 01_2, 10_2, 11_2). Figure 3.3 shows possible clock configurations.

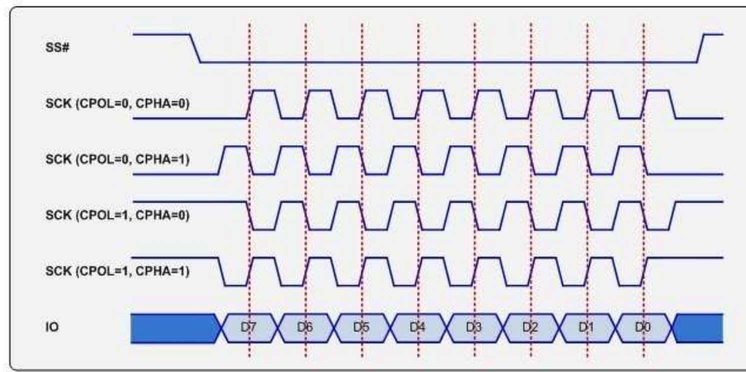

Figure 3.2: Possible clock configuration.[23]

3.4 Generic Timer Module

The Generic Timer Module (GTM) is a device developed from Bosch for automotive field used as coprocessor and capable to do some complex I/O operations without CPU interaction. It consists of a series of submodules designed to do different works independently from each other or from the CPU, like generate clock signal, sending output signals and capture input signals.

The GTM device is capable of handling parallel data transmission, this is an important feature when is necessary to communicate with many devices.

Some submodules that compose the GTM device are: ARU, ATOM, TOM, TIM and the CMU.

GTM device contains, especially for some submodules like ATOM, TOM, TIM and CMU, more than one unity of those modules, and each of them is divided into channels. So for instance, on GTM device there are 4 ATOM modules, each one is divided into 8 channels, another example is the CMU module, divided into 3 different submodules: Fixed Clock Generation (FXU), External Clock Unit (ECU), or Configurable Clock Generation (CFGU), with different number of channels for each submodule.

To manage the transmission of the ATPG patterns, not all GTM submodules will be used (and therefore discussed in this thesis), the used submodules are: ARU, ATOM, CMU, TIM, PSM and the MCS.

3.4.1 Advanced Routing Unit (ARU)

The Advanced Routing Unit (ARU) is the submodule responsible of data routing through the GTM submodules. It connects a data source to a data destination, this connection is called data stream, and it uses interrupts to signal when data is ready to be read or to be written. In the GTM device, every data source has its unique address so it must be specified to the ARU when a read or write request is done by a data destination (for instance the ATOM that is a data destination can read data from the PSM, which is a data source, by specifying its address to the ARU). Every read request is destructive, so is not suitable to connect more than one data destination to one data source. If same data must reach more than one data destination, the broadcast (BRC) module must be used.[24]

Each data word transferred between the ARU and the various submodules of the GTM must be 53-bits wide: bits from 52 to 48 are called ARU Control Bits (ACB) with special meaning depending on submodule that will receive them, bits from 47 to 24 are the first data word called high data word, and bits from 23 to 0 are second data word called low data word.

ARU routes data using a round-robin algorithm, so data requests are served one at a time following a specific ordering.

ARU module is provided with some registers too and the most important are: ARU_ACCESS register, used to set a read or write request, ARU_DATA_H that contains the bits of ARU data word from bit 24 to bit 52 and ARU_DATA_L that contains bits of ARU data word from bit 0 to 23.[24]

3.4.2 Clock Management Unit (CMU)

The Clock Management Unit (CMU) is responsible of clock generation for the GTM submodules, it consists of three submodules: Configurable Clock Generation (CFGU) that provides clock signals to TOM, TIM, ATOM submodules, the External Clock Generation (ECU), not used here, which can generate clock on the output and the Fixed Clock Generation Unit (FXU), not used here, where every channel of this submodule provides a different fixed clock frequency.[24]

3.4.3 Parameter Storage Module (PSM)

Parameter Storage Module (PSM) is used to let data to be stored somewhere before being routed by the ARU to a data destination or being passed to the CPU. The PSM is organized as a FIFO structure, divided into three submodules called interfaces: first one is the AEI to FIFO data interface (AFD), used by the CPU to fill the FIFO with data through the AEI bus (same interface is used by the CPU to read data into the FIFO too), second one is the FIFO to ARU data interface (F2A) used by the ARU to pass data through the FIFO and the submodules and the last one interface is the FIFO itself.

The FIFO is divided into channels named streams and is possible to treat every stream as an independent FIFO in order to have better data organization. Each FIFO may contain a maximum number of 1000 words of 29 bits each, so data from the ARU (53 bits wide) must be divided into 2 words of 29 bits where the first 24 bits are the data bits, the other 5 bits are the ACB bits.

The FIFO can operate in normal mode or ring buffer mode, is also possible to configure the FIFO depth (not greater than 1000 words) to optimize the resources request.[24]

3.4.4 ARU connected TOM (ATOM) submodule

ARU connected TOM (ATOM) module is, like the name says, a submodule where a Timer Output Module (TOM), used to generate output signals, is directly connected to the ARU submodule that will provide signals to transmit. This modules is similar to a TOM module but with some differences: first every ATOM module contains only 8 channels, second every ATOM module contains an ATOM Global Control Unit that manages the channels (this module let possible to configure each channel of the used ATOM module).

Every ATOM channel can be configured to work in one of four different modes: Signal Output Mode Immediate (SOMI), Signal Output Mode Compare (SOMC), Signal Output Mode PWM (SOMP), and Signal Output Mode Serial (SOMS).

For the purpose of this thesis, discussed on chapter 8.7.1, the used working mode is the Signal Output Mode Serial (SOMS), which allows to the chosen ATOM channel to behave like a shift register in output and shift out the data passed by ARU unit.
When the used ATOM channel works in this mode, the data source is defined at system startup, the chosen source is the FIFO where the CPU will put data to transmit and where the ARU will take data to pass to ATOM when required.[24]

3.4.5 Timer Input Module (TIM)

Timer Input Module (TIM) is used to filter and to capture input signals. For the ATPG module, this device is used to capture signals coming from the DUT during data shifting.
The captured data will be saved into the FIFO and, when the capture process is complete, received data will be passed from the GTM device to the CPU.
The TIM module can capture input signals in different ways, depending by the capture source: it can be an interrupt, or an auxiliary clock source and so on.
The data captured from the TIM module can be passed to the CPU directly by using the AEI interface (which connect the CPU to the GTM device) or it can be passed to another module using the ARU.[24]

3.4.6 Multi-Channel Sequencer (MCS)

The Multi-Channel Sequencer (MCS) is a submodule capable of generic data processing, and it is connected to the ARU.
This submodule is mainly used to automate the data transmission or to perform extended data processing of input data resulting from TIM module.
The MCS is provided with its own assembly language composed by a few instructions, so this submodule is configured for the use by writing the assembly code of its task.
This module is composed of eight channels that can act as a parallel tasks managed by a scheduling algorithm and every channel is provided with a set of registers,

numbered from R0 to R7, where R0..R5 and R7 are used as a general purpose register meanwhile R6 is an index register used by the ARU to know from where data to route will be read or to allow the MCS channel itself to write or read to an address of the MCS RAM memory. Other two registers, named Set Trigger Register (STRG) and Clear Trigger Register (CTRG) are shared between the eight channels of the same MCS module, so a channel can trigger another channel of the same MCS submodule.

The MCS submodule embeds a single data path composed of five pipeline stages, this data path is shared between the eight channels and each channel can execute its own microprogram that is stored in the MCS RAM memory.

Every channel can be triggered by the CPU by setting the corresponding bit inside the STRG.[24]

3.5 Hardware configuration for the implemented modules

The configuration of the SPI and the GTM devices are:

- Two different SPIs are used: first one acts as a master SPI and gives the clock signal to the DUT and to the second SPI. This SPI is used to send data on TMS line and, at the same time, receive data from TDO line saving it into the TMS buffer (to save resources). The second SPI acts as slave SPI, used only to transmit (without receiving anything) data on TDI line contained into the TDI buffer. The master SPI CS line is connected to slave SPI CS line, thus master will enable slave SPI when data must be transmitted.
- Transmission clock frequency is 1MHz.
- Clock phase and clock polarity are both set to 1 (11_2), so a bit is sampled during the raising edge and shifted into register during the falling edge meanwhile the idle state of the clock line is high.
- Shift direction is LSB (Least Significant Bit).

GTM device configuration is the following:

- One ATOM module with two enabled channels configured to work both on SOMS mode.
- One Timer Input Module (TIM) used to capture input bits from DUT.
- FIFO enabled in order to store test patterns to transmit and store captured data words coming from TIM module before passing it to the CPU.
- ARU enabled in order to route data from the FIFO to the ATOM module and from the TIM module to the FIFO.
- LSB shifting direction.
- The MCS is used to manage data transmission: one MCS channel manages the data transmission, another one manages the clock transmission. The used scheduling algorithm is the round robin scheduling.

The GTM configuration will be discussed in detail on the chapter 8.7.

CHAPTER 4

Real Time Operating System (RTOS)

4.1 Chapter introduction

The entire software for the BurnIn+SLT is composed by a set of functionalities to let the supervisor to communicate with the DUT, with the terminal or read and collect data concerning the test. The software to write is complex, so the traditional code architecture for microcontrollers, named *bare-metal,* is no longer sufficient. In this case, and in all cases where the code reach a high complexity level, the best way to write good code and ensure some requirements like the determinism of the code execution or the scalability is to use an RTOS. This chapter will show the differences between the traditional way to write a code for the microcontroller and the usage of an RTOS, and what advantages brings the use of it meanwhile, on last paragraphs will be discussed the use of FreeRTOS.

4.2 Defining RTOS and advantages on using it.

For embedded systems, depending on the system complexity, the code can be written with two different architectures: first one is named *bare-metal* architecture, the second is the one that provides the use of an RTOS.

- Bare-Metal: on *bare-metal* programming, which is widely used for its simplicity, the entire code is encapsulated inside an infinite loop whose purpose is to keep the program running until the microcontroller is powered and where all peripherals are managed following the write ordering. An interrupts set is used to let the microcontroller to become aware of some event by a device or sensor so it can therefore immediately serve the request. Bare-metal programming is simple, quick, it requires not much resources and it can be easily debugged, but on the other hand, with growing of the complexity of the system, its management became more difficult and it is

not anymore deterministic, in addiction future changes to the code are difficult to do and generally unsafe.

- RTOS: a Real Time Operating System (RTOS) is a system that allows the management of various peripherals through the use of tasks. The benefits of an RTOS are mirror-like to those of bare-metal programming, so more resources are required, system speed is reduced and is more difficult to debug (apposite tools are necessary to debug a system that uses an RTOS), on the other hand, a system that makes use of tasks is totally deterministic, the code of each peripheral can be managed more easily because usually one task is assigned to exactly one peripheral and every task is independent from the other ones. At least is easier to add code and the interrupts can be managed more efficiently thanks to a task priority system and to a scheduling algorithm, which decides what task should be executed every time. If two tasks must communicate, is possible to use some structures like queues, is possible to manage concurrent access to some variables or to the peripherals using semaphores or mutexes.

	Bare-Metal	RTOS
Resources hungry	NO	YES
System speed	YES	NO
Easy to debug	YES	NO
Useful when a lot of peripherals must be managed	NO	YES
Useful with sophisticated user interface	NO	YES
Useful to manage interrupts	NO	YES
Easy management for system updates	NO	YES

Table 4.0: Differences between bare-metal and RTOS

4.3 RTOS concepts

An RTOS is something intended to reduce buffering delays and to process like real-time application the data when it comes in.
Key factors in an RTOS are how quickly or more predictably it can respond.
An RTOS has the following goals:

- Optimizing code development: with the growing of the system complexity, the ROTS allows to divide the entire code into tasks that can be developed from different programmers.
- Synchronization: In a small system, synchronization is ensured by using global variables. With system grow, there is more change to being corrupted by incorrect peripheral synchronization. The use of an RTOS ensures a better synchronization of the tasks without any corruption problem.
- Timing: all the RTOS provide time management functions that can be used to achieve task delay, timer handling and so on without understanding the hardware mechanism.[25]

Every RTOS is composed by two entities: the scheduler, which manages tasks execution, and the tasks itself. The scheduler manages the tasks using one of the scheduling algorithm, which can be set at system startup. The implemented scheduling algorithms can be:

- Preemptive scheduling: this algorithm allows task execution following a priority ordering. Task in execution stays in this state until the CPU does not receive an interrupt or until a task with a major priority level is created. In both cases, the task in execution goes on pause, the control passes to the newly created task until its execution is done, until it is not blocked or until a task with an even major priority is created.
- Non-preemptive scheduling: if this algorithm is used, the scheduler can only start a new task. Created task stays on execution until it does not give control to another task or until it is not blocked by something, for instance an interrupt.
- Time-slicing scheduling: if this algorithm is used then the scheduler will give a time slice named CPU time, which is set at system startup, to every task. When CPU time for a task finishes, CPU control goes to another task. This algorithm is not suited for real time application, mainly because tasks are not managed following a priority ordering. Often, if a time-slicing scheduling is used, the task goes on execution following a round-robin ordering. [26]

A task is a block of code, independent from other tasks, with a memory space called Task Control Block (TCB) used to have trace about the task variables and the context itself.

Every RTOS allows user to create three different types of task:

- Task: a task generated during normal code execution, normally these tasks are the ones that manage the code execution and have longest life time.
- Interrupt Service Routine (ISR): a task of this type is created when an interrupt is launched. This task stays in execution for all the time needed to manage the interrupt and it is destroyed at the end of interrupt management.

- Idle Task: this task is never used, it exists only to let the scheduler to stay alive when no one task is in execution.

Task activity is identified by its state. States are grouped into two groups:

- Running: a task in running state is the one that have control over the CPU. Only one task a time can be in running state.
- Not running: not running is a state formed by many sub-states. Every task that not executing is in not running state.

Not running sub-states are:

- Blocked state: tasks in blocked state are waiting for some event to occur to pass from blocked state to ready state.
- Suspended state: tasks in suspended state are not available to the scheduler. A task cannot go on the suspended state by itself, the transition from any state to the suspended state is handled by the code of the other tasks.
- Ready state: tasks in ready state are the ones waiting to be executed. Scheduling algorithm choose only one from all tasks that are in ready state.[27]

Idle task is always in ready state and it cannot be blocked.

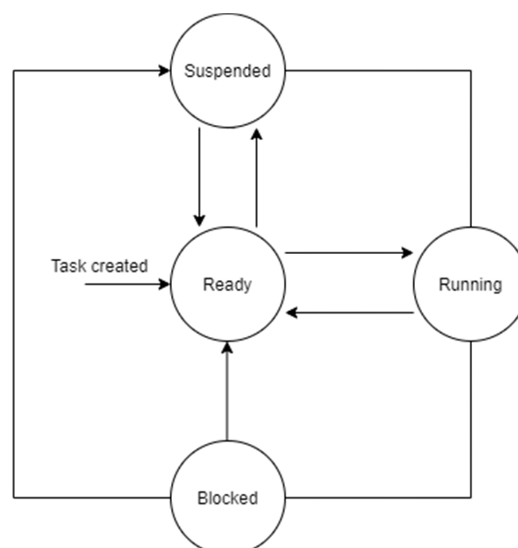

Figure 4.0: Task states.

4.4 FreeRTOS

To add the RTOS on the BurnIn+SLT platform, FreeRTOS is used. This RTOS was chosen mainly because it is freely downloadable from the FreeRTOS site https://www.freertos.org/a00104.html.

FreeRTOS can be configured from the configuration header (.h) file called *FreeRTOSConfig.h*. Here is possible, by setting the proper macro values, to configure the RTOS in order to determine some aspects like the scheduling algorithm, the memory management algorithm, maximum priority level and so on.

FreeRTOS is composed by a scheduler, the tasks and a set of functionalities used to manage the tasks. Tasks can communicate each other using some structures like the queue and synchronize themselves using structures like semaphores and mutexes.

FreeRTOS scheduler uses the preemptive or the time slicing algorithm to manage tasks execution, in case of time slicing scheduling algorithm the time slice duration can be configured from the *FreeRTOSConfig.h* file by setting the *configTICK_RATE_HZ* macro value.

To adapt FreeRTOS to the used microcontroller, is possible to choose the memory management algorithm. FreeRTOS offer five algorithms, they are:

- Heap-1: the memory is divided into arrays and divided between the created tasks. Heap-1 do not allows memory to be freed.
- Heap-2: unlike Heap-1 allows memory to be freed and it uses a best fit algorithm to allocate memory.
- Heap-3: uses system *malloc()* and *free()* functions, so the linker defines the size of the heap.
- Heap-4: works by subdividing an array to smaller blocks, statically declared, so it will make the application to consume a lot of RAM, even before any memory has actually been allocated from the array. Heap-4 use a best fit algorithm, but unlike heap-2, it combines adjacent free blocks of memory.

- Heap-5: it uses an algorithm that can allocate and free memory identically to heap_4 but, unlike the latter, heap_5 is not limited to allocate memory from a single statically declared array; heap_5 can allocate memory from multiple and separated memory spaces.[28]

After configuring FreeRTOS, the scheduler will start only when *vStartScheduler()* function is reached during code execution. The first task that goes on running state must be created before this function is called.
A task can be created by calling the FreeRTOS function *xTaskCreate*, this function takes as parameters:

- A pointer to the function to be executed by the task.
- A character array representing the task name. This parameter can be NULL, it is used only in order to recognize the created task.
- An integer representing the task stack size.
- A pointer to a structure containing task function parameters. The tasks that use the JTAG and ATPG modules do not take any parameters.
- An integer representing task priority value.
- A pointer of defined type *TaskHandle_t* that is used to point to the created task when is necessary (for instance, if a task want to delete another task, it must call *vTaskDelete* function passing as parameter the pointer to the task to delete).

Depending on chosen scheduling algorithm, a task can go on running state after being created.

4.5 Tasks communication ways

An important aspect of every RTOS is the possibility to let tasks to communicate each other, when two or more tasks exchange data, is said that they are communicating. To allow tasks communication, FreeRTOS implements some mechanisms like queues or notification. The tasks that make use of JTAG and ATPG modules communicate with the use of the notification system.

Communication doesn't never occur between ATPG task and JTAG task, they communicate only with a third task listening on a Universal Asynchronous Receiver Transmitter (UART) peripheral, which is used to let the supervisor microcontroller to receive commands from a terminal.

Notification system consists on sending of a specific value from a task to another one that must be on blocked state waiting for a notification, a notification can be sent from a task to another one by calling *xTaskNotify* function. This function takes three parameters:

- A handler to the task to notify.
- The notification value. It is a 32-bit value used to notify the task whose receives it.
- The action to do when the notification value is sent. It can be one of those: *eNoAction* if the value should not be modified, *eSetBits* if the receiving task notification value should be bitwise OR'ed with the value passed as parameter in the *xTaskNotifyWait* function, *eIncrement* if the notification value received from the task should be incremented, *eWriteValueOverwrite* that allows to a new notification value to overwrite a pending notification value if the task that received it has not read it yet and *eWriteValueWhitoutOverwrite* which will not overwrite a pending notification value if the task that received it has not read it yet (in this case the newly notification value will be lost).[28]

A task that should receive a notification must be on blocked state waiting for a notification. To let task go on blocked state waiting for notification it must call the function *xTaskWaitForNotification*. This function takes 4 parameters:

- *UlBitsToClearOnEntry*: If the calling task did not have a notification pending before it called *xTaskNotifyWait*, then any bits set in *ulBitsToClearOnEntry* will be cleared in the task notification value on entry to the function.
- *ulBitsToClearOnExit*: If the calling task exits from the function *xTaskNotifyWait* because it received a notification or because it already had a notification pending when *xTaskNotifyWait* was called, then any bits set

in *ulBitsToClearOnExit* will be cleared in the task notification value before the task exits the *xTaskNotifyWait* function.

- *pulNotificationValue*: is the pointer pointing to the address where the received value will be saved.
- *xTicksToWait*: the number of ticks that a task should wait before exit from this function. If the parameter is the macro *portMAX_DELAY* then the task will wait indefinitely.[28]

The advantage of using the notification system is that the notification value can also be the memory address of a variable. In the specific case of JTAG and ATPG tasks, the notification value is the memory address of a structure that contains all required information to execute the command passed from the task listening on the terminal, on other hand the disadvantage of the notification system is the fact that however communication can take place only between two tasks but, for the implemented modules, does not arise because each command is addressed only to the JTAG task or to the ATPG task, never both simultaneously.

4.6 The FreeRTOS CLI

In order to put in place an effective iteration with the MCU, a console with a command line interpreter must be defined. FreeRTOS offer, as an additional module, a Command Line Interpreter (CLI) that can be used to define a series of commands to be executed. It is a freeware downloadable from the FreeRTOS website.

34.6.1 Commands architecture

To create a command, first a structure of the type *CLI_Command_Definition_t* must be created, it represents the command definition. This structure contains the following fields:

- A string representing command name.
- A string representing command description.
- A function pointer pointing to the function to execute when a command is correctly inserted.
- An integer representing the number of expected parameters.

FreeRTOS saves user defined commands inside a linked list. A linked list is a linear data structure where each element is a separate object. The elements of a linked list, named nodes, are not stored in contiguous location of memory but a node is linked with the next one by a pointer that stores the address of the next node.
In order to let a command to be callable, it must be registered by calling FreeRTOS function *FreeRTOS_CLIRegisterCommand*, in this way the command is inserted inside the linked list as last element. This function must be called at system start-up and for every defined command in order to register it.

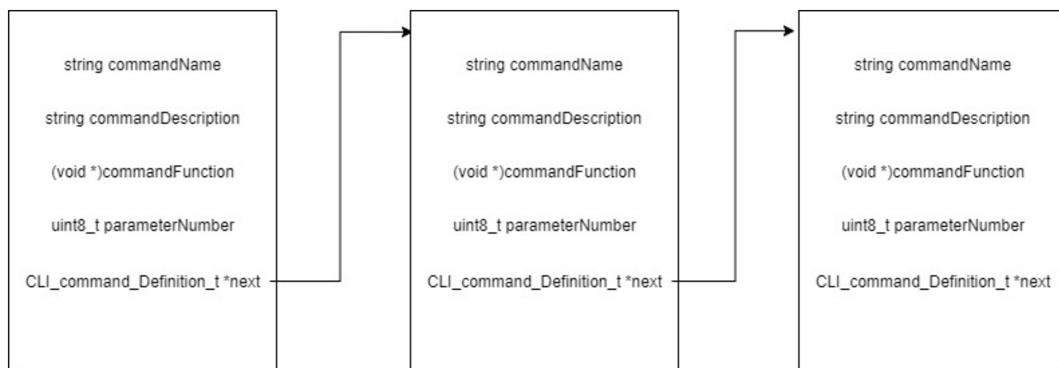

Figure 4.1: Commands linked list example

Command parsing is done by calling *FreeRTOS_CLIProcessCommand* function. This function takes as parameters the input buffer containing input string to parse, a buffer that will contain the string to send to output and the size of output buffer.
Every input command in order to correctly be parsed must be written with following syntax:

<command name> <parameter_1> <parameter_2>….<parameter_n>

When *FreeRTOS_CLIProcessCommand* function is called, it will start to parse the command string.
The first command word, which is the command name, is compared with command name string of each node of the linked list containing the implemented commands, if the command name is found then the number of inserted parameters is compared with the *parameterNumber* field. If the values are equal then the command is executed calling the function pointed by the function pointer. If some comparison does not match, then an error is reported.
The function associated to the command must have the following signature:

BaseType_t functionName(uint8_t *output, size_t len, const uint8_t *input)

BaseType_t is a FreeRTOS defined type, *output* is a pointer to string representing any message to send in output, *len* is an integer meaning the size of the output string, and finally *input* is a pointer to the input string.
When the function associated to the command is called, it receives the input string as parameter, is up of the user to parse again the input string because it will be passed in full.
To retrieve a parameter from command input string, the function *FreeRTOS_CLIGetParameter* must be called passing as parameter the command input string and the positional value of the wanted parameter.

```
param = FreeRTOS_CLIGetParameter((char *)input, nParam, &param_len);

if(param != NULL){

    //First and only parameter is bits number

    snprintf(buff, param_len+1, "%s", param);
    sscanf(buff, "%d", &tDataStruct.bits);

    //check if bytes of transaction is less than buffer dimension
    if((bits / 8) > JTAG_BUFFER_SIZE){

        hjtag.status = DATA_OVERFLOW;

        strcpy((char *)output, errorTable[hjtag.status - 1]);
        return pdFALSE;
    }

    //Set tDataStruct ID to JTAG_READ and give it jtagTask
    tDataStruct.ID = JTAG_READ;

    xTaskNotify(jtagTask, (uint32_t)&tDataStruct, eSetValueWithOverwrite);

    //Wait until notify is givwn by jtagTask
    xTaskNotifyWait(0x00, 0x0, NULL, portMAX_DELAY);

    strcpy((char *)output, "Readed data: ");
    strcat((char *)output, hjtag.lld_jtagS.rcvStr);
    strcat((char *)output, "\r\n");
```

Figure 4.2: Command function example

4.6.2 FreeRTOS CLI integration

FreeRTOS CLI is used to let the user to communicate with the microcontroller from a terminal.

To do this, the serial console based on UART was exploited with following configuration:

- 38400 baud
- 8 bits frame
- 1 stop bit
- Asynchronous mode

No DMA or interrupts are used to manage the UART transmission.

CLI console works on the following way: it will wait for a character a time from the UART peripheral, check if it a special one, like carriage return (\r) or backspace (\b) or new line (\n) (which is simply ignored) and when the input is totally received, then the *FreeRTOs_CLIProcessCommand* function is called in order to parse the command. When command execution is over if there is a message to send in output then the task controlling the CLI will wrote it on UART port and start again to listen for a character.
If a special command, named *send_pattern* is received, then the CLI task will not anymore use FreeRTOS CLI functions, but will send the data to the ATPG task.

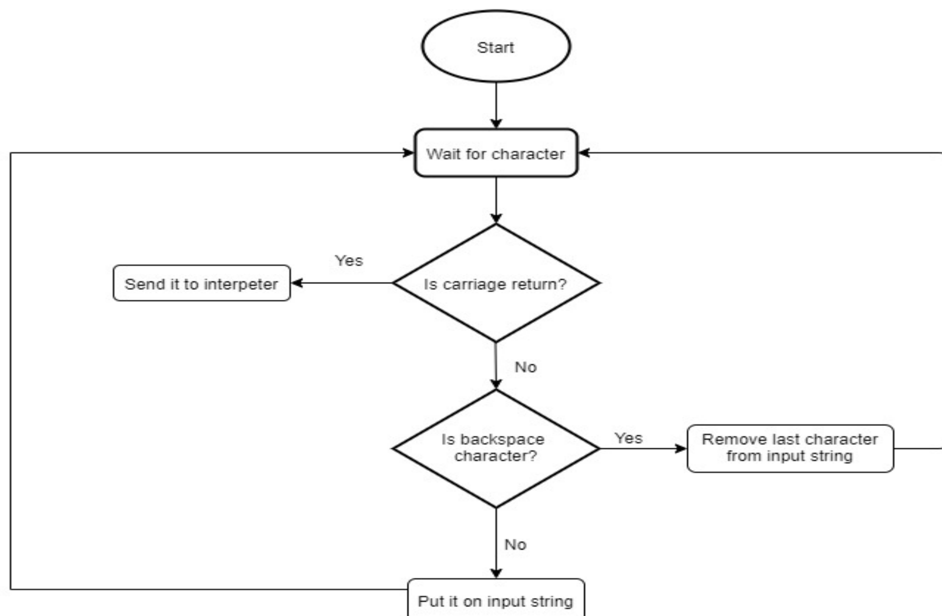

Figure 4.3: CLI flow

Commands that user can send over CLI to use the JTAG and ATPG modules are:

- enable_jtag: create JTAG task.
- disable_jtag: delete JTAG task.
- jtag_transaction: do a JTAG transaction.
- jtag_reset: send reset command to DUT.
- jtag_read: read n bits from a data register.
- jtag_cmpData: compare received data on last JTAG transaction with the data passed as parameter.

- send_pattern: switch the CLI from JTAG to ATPG and send n bytes of data over ATPG line.

The communication with terminal is managed by a task named *CLITask*. The task is created at system startup with a call to the function *taskInit,* and it is never deleted.

Chapter 5

Programming software

5.1 Chapter introduction

A microcontroller is composed by a set of peripherals that require to be configured before being used or require a specific set of functionalities to be used.
To avoid to repeat the configuration process every time or to avoid to implement every time all the necessary code to manage the peripherals, usually the engineers uses some specific tools to write the microcontroller code, to configure it and to debug it.
This chapter will briefly discuss the used software to write the code for the SPC5 microcontroller family and for the STM microcontroller family.

5.2 Programming a microcontroller

A microcontroller works executing the code written on its memory. Is important, during the development of a system, to choose the best programming language to use. Before to choose, is mandatory to consider the fact that a microcontroller have limited resources so the used programming language must fit some requirements:

- It should be a compiled language, in this way when it is executed does not require additional software and it will use less amount of memory.
- It must be a low level language in order to ensure full control over all the aspects of the microcontroller, in addition, a low level programming language has reduced execution time and requires less memory.

The programming language that better fit these characteristics, and which is used to program almost every type of microcontroller, is the C language or the C++ language.

The code to be executed is written using apposite software: one is used to write code for STM microcontroller family, another for SPC5 microcontroller family. Used softwares are CubeIDE (for STM microcontroller family), freely downloadable from the site https://www.st.com/en/development-tools/stm32cubeide.html and SPC5Studio for SPC5 microcontroller family, also freely downloadable from the site https://www.st.com/en/development-tools/spc5-studio.html.

Both software are based on Eclipse platform, SPC5Studio uses the Plug-In Development Environment (PDE) to develop plugins that can be added to the IDE.[29]

Both tools are composed by three pieces: the configurator, the IDE itself and the code debugger.

In addiction SPC5Studio generates code compliant with MISRA 2012 directives but they are not followed during the implementation of the modules.

5.3 The configurator

The configurator is the tool that the user, through a GUI, can use to configure the microcontroller by setting the pins to use for I/O, enable the peripherals to use (as SPI, UART and so on), choose working clock frequency and so on.

Microcontroller configuration phase ends with the generation of the code needed to set the microcontroller, this code can be written all on the *main.c* file, or defined into separated files, often one file for each used peripheral.

Code written by user must be placed after the call to the configuration functions, this in order to ensure the correct functioning of peripherals and pins. The generated code includes also low-level libraries that contain the functions necessary to use a specific device, for instance: if an SPI peripheral is enabled, then the generated code will have the functions to transmit and to receive, while if it disabled, these functions will not be available and a call to them will result in a compilation error. The use of the configurator is advantageous since it frees the developer from the process of configuration of the microcontroller, letting them to concentrate on the

system code, in addition, the generated code changes every time the configuration of the microcontroller changes too.

Figure 5.0 CubeIDE and SPC5Studio configurators

5.4 The IDE

Once the microcontroller has been configured, the IDE shows *main.c* file, where the user code can be written and the user made libraries can be included.
CubeIDE and SPC5Studio structure the main in different ways: CubeIDE with the usage of comments separates the parts where the user code can be written with the ones that are automatically generated, in this way when the microcontroller is reconfigured, all the code but the parts reserved for the user will be recreated. In the case of SPC5Studio, the configuration is done by calling specific functions for each module, they are inserted into a single function named *conponentsInit* and future changes to the configuration of the microcontroller will change the calls to those functions within this function, so the main is never modified or regenerated.
A project generated with both of these two software have a tree architecture, where at the top level is the project folder, inside it there are folders containing the libraries of the used peripherals (CubeIDE includes in the project only the code of the enabled peripheral, meanwhile SPC5Studio includes the code of all the peripherals, but with the use of macros only the code of the enabled ones is compiled). It is also possible to add other folders to the project, ensuring a certain order inside the project. If SPC5Studio is used, is necessary to modify the *user.mak* to let the compiler to compile files that are not inside the *source* folder.

5.5 The debugger

When the code is written, it will be compiled and debugged. To debug the code, CubeIDE incorporates a debugger, while SPC5Studio relies on the external debugger named Universal Debug Engine Software Development Kit (UDE-STK), downloadable from the site (https://www.pls-mc.com/universal-debug-engine-ude-and-microcontroller-debugger-for-aurix-tricore-power-architecture-cortex-arm-xe166xc2000-xscale-rh850-sh-2a_c166st10-stm32-stellar-s32v234-s32/universal_debug_engine-a-802.html).
During the debugging phase, is possible to monitor step by step the code execution in order to find possible bugs to fix. It is also possible to read the value of a variable

or the content of a peripheral register and modify it and is possible too to add a code breakpoint in order to pause code execution at certain point.

A last tool made available by CubeIDE, named CubeProgrammer, allows the effective loading of the code inside the memory of the microcontroller, so as to start the execution immediately after powering it.

Chapter 6

Software Implementation

6.1 Chapter introduction

The developed modules must respect the software architecture designed for the BurnIn+SLT platform. To achieve this goal, the development process took place by paying attention to the aspects of versatility and consumption of resources, both from the hardware side, trying to find which devices are best suited to transmit data, and from the software side with the development of a code that meets three criteria:

- Modularity: for easy handling, the modules are divided into layers. The lower layer is composed by the set of utilities for data transmission and reception, the intermediate level instead includes the features for the preparation of the buffers to transmit. Finally, the highest level is the one used to interface the user or other modules with the lower levels.
- Structure: the modules must have a structure that ensures their easy control and easy management. To achieve this, some data structures were defined: the former, which is a low level driver structure, contains the buffers and the variables used to manage data transmission, the latter, the higher level structure acts as a wrapper to the lower one and contains the flags used to give information to the user about driver status.
- Portability: modules should be easily imported and adapted to other projects, so they are written trying to make them less dependent from the microcontroller that uses them. The division into layers brings advantages in this area, as only in the lower level are made explicit reference to the functionality of the microcontroller.

This chapter will show the architecture of the implemented modules before discuss them in details in the chapter 7 and chapter 8.

6.2 Targets

The work of this thesis was targeted either for SPC58EC (automotive) or STM32 (consumer) microcontroller; the latter has been exploited whenever necessary in order to verify the developed algorithms with a friendlier platform without licensing limitation and code restriction, while the former has the same peripherals as the target microcontrollers.

The SPC5EC microcontroller have the following characteristics:

- High performance e200z420n3 dual core.
- 4224 KB (4096 KB code flash + 128 KB data flash) on-chip flash memory: supports read during program and erase operations, and multiple blocks allowing EEPROM emulation.
- 176 KB HSM dedicated flash memory (144 KB code + 32 KB data).
- 384 KB on-chip general-purpose SRAM (in addition to 128 KB core local data RAM: 64 KB included in each CPU).
- Multi-channel direct memory access controller (eDMA) with 64 channels
- 1 interrupt controller (INTC).
- Crossbar switch architecture for concurrent access to peripherals, Flash, or RAM from multiple bus masters with end-to-end ECC.
- Boot assist Flash (BAF) supports factory programming using a serial bootload through the asynchronous CAN or LIN/UART.
- Junction temperature range: -40°C to 150°C.[6]

The price of this microcontroller is about 14$.

The STM32 family are consumer oriented microcontroller, in particular the one used to make the porting of the modules was the STM32F446RE microcontroller: a user oriented microcontroller with following characteristics:

- Core: Arm® 32-bit Cortex®-M4 CPU with FPU, frequency up to 180 MHz.
- 512 Kbytes of Flash memory.
- 128 Kbytes of SRAM.
- Dual mode QuadSPI interface.
- 1.7 V to 3.6 V application supply and I/Os.

- General-purpose DMA: 16-stream DMA controller with FIFOs and burst support.
- SWD and JTAG interfaces.
- SPDIF-Rx.
- Up to 4 × I2C interfaces (SMBus/PMBus).
- Up to four USARTs and two UARTs (11.25 Mbit/s, ISO7816 interface, LIN, IrDA, modem control).
- Up to four SPIs (45 Mbits/s), three with muxed I2S for audio class accuracy via internal audio PLL or external clock.[7]

The price of this microcontroller is about 3.6$.

The boards are provided with ST-LINK programmer that directly communicates with the microcontroller, in order to upload the code and debug it.

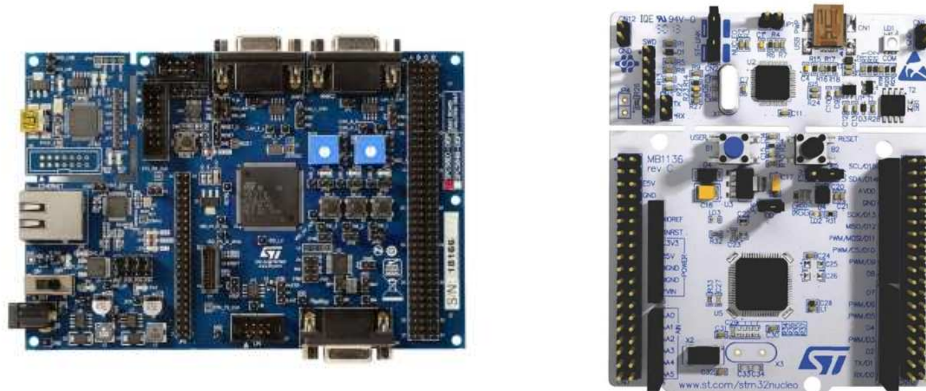

Figure 6.0: SPC58EC-DISP board and STM32F446RE board

6.3 Modules architecture and configuration

Before starting to use the modules they must be configured in order to work properly. The configuration is done by setting the value of some macros on the *cnf.h* header file and these are used to refer to the used peripherals and their configuration during code execution.

In the *cnf.h* file is possible to set the handler to the SPI master and SPI slave, assign the configuration of these two SPIs to two apposite macros and configure the dimension of data buffers to adapt them to the available resources.

If a terminal is used, is possible too to assign the used peripheral to refer to it during code writing. In order to ensure the correct configuration of the modules, a file named *errorCheck.h* is added in order to check at compile time if the macros are defined with the correct value. If some macro is not defined, an error is issued and the compilation process is stopped (the check is not done on the type of peripheral assigned to the macro).

Looking into the modules architecture, they are divided into layers in order to ensure better code management.

Every module is divided into three layers: starting from above, the first layer is a wrapper mainly composed by these functions that user must use in order to transmit JTAG data and instructions or to manage the moving of the ATPG patters. The second layer, named transport layer, is composed by the functions that work on data buffers, filling them correctly with data that come from a command or from an explicit call of some function from an upper layer, before passing the filled buffers to the lower layer that is the third layer, the lowest one, where them are transmitted using the SPI or another peripheral. The functions that compose the lower layer are wrapper functions for the ones implemented into the SPC5Studio libraries that manages the used peripheral. The functions from lower layer are used in order to initialize the modules at system startup and transmit data, and all but two are declared as static functions. To ensure maximum modularity, the modules are used by two different tasks: the task that uses the JTAG module functions, created when *enable_jtag* command is sent to the microcontroller and is destroyed by sending the *disable_jtag* command, and the task that uses ATPG module functions, created

when *send_pattern* command is sent to the microcontroller and it is destroyed when the command execution is complete.

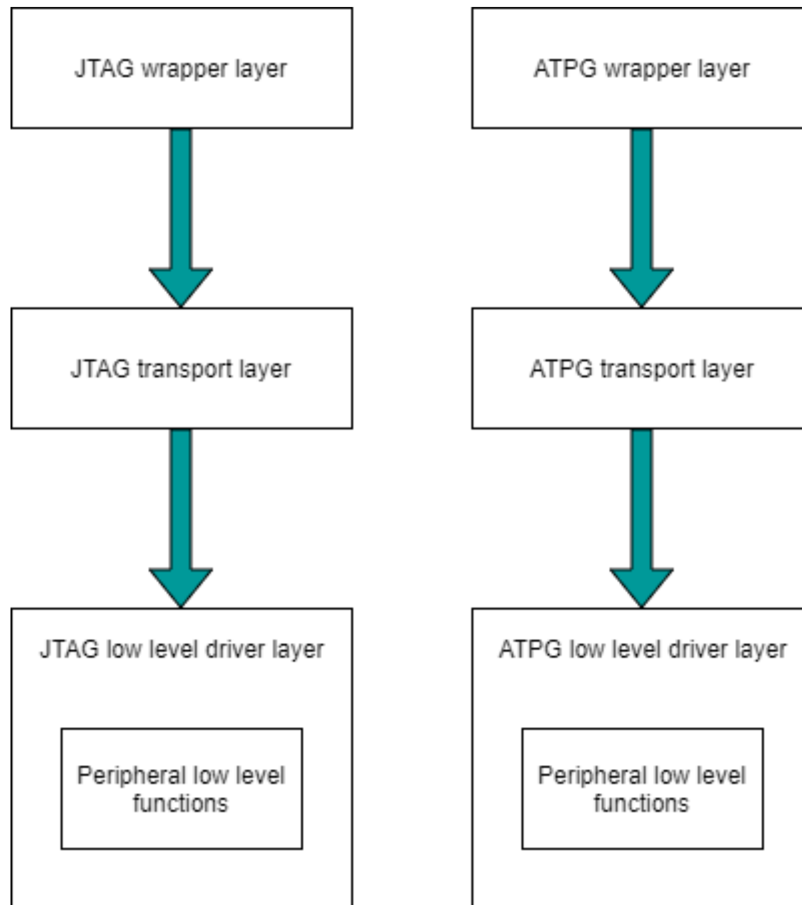

Figure 3.1: Modules architecture

6.4 FreeRTOS integration

The modules are used by two separate tasks that communicate with the one listening on UART in order to execute the commands that come from the terminal.
The tasks are independent each other and they are created only when is necessary to reduce resource demand.

6.4.1 Tasks priority

The used scheduling algorithm is preemptive, so the execution order is given by the task priority value, due to this policy the JTAG task and ATPG task have a higher priority level than the CLI task, so they go on running state after being created and, soon after, they go on blocked state waiting for a command to execute or data to transmit leaving the CPU to the CLI task again.
The priority value is the same both for the JTAG task and the ATPG task and it is given by the *tskIDLE_PRIORITY* value augmented by 2, meanwhile CLI task has a priority level given by *tskIDLE_PRIORITY* value augmented by 1. The value of the *tskIDLE_PRIORITY* macro can be changed, but for the implemented modules it is not modified and it is equal to 0.

Task		Priority
ATPG Task	JTAG task	tskIDLE_PRIORITY +2
CLI Task		tskIDLE_PRIORITY +1
Idle Task		tskIDLE_PRIORITY

Figure 6.2: Task priorities order

6.4.2 Tasks communication

The tasks communicate with use of the notification system(see 4.5).
As mentioned before, the JTAG task and the ATPG task communicate only with the CLI task, never between each other. As notification value the address of a defined type structure named *taskData_t* is used.
This structure contains the following fields:

- *ID*: a flag that holds the command ID used by the JTAG task to recognize what function should be called.
- *Bits*: field holding the bits number to send over JTAG.
- *Byte*: field holding the bytes number to send over ATPG.
- *currState*: a flag holding the current state of the JTAG state machine.
- *firstState*: flag holding the first state that the JTAG state machine should reach. This flag is read only if a JTAG transaction command is inserted.
- *nxtState*: flag holding next state that JTAG state machine should reach. This flag is read only if a JTAG command is inserted.
- *txBuffer*: buffer containing data to send over JTAG or ATPG.
- *rxBuffer*: buffer containing received data from JTAG or ATPG.
- *cns*: flag holding what type of console is used. It can be *ATPG_CONSOLE* or *JTAG_CONSOLE*. The value of this field change when the microcontroller receives the command *send_pattern*.

Only one structure of this type is created then it is shared between tasks with a pointer declared inside every task function.

6.5 Low level driver functions

Depending on the used microcontroller, data transmission is managed using the low level APIs for the used peripheral.
For Chorus4M board, and for each microcontroller of the SPC5 family, the used low level APIs come from SPC5Studio software and they are:

- *spi_lld_init*.
- *spi_lld_start*.
- *spi_lld_stop*.
- *spi_lld_send*.
- *spi_lld_exchange*.
- *serial_lld_start* (for UART peripheral).

Spi_lld_init, *spi_lld_start* and *spi_lld_stop* are used to initialize the SPI peripheral, to configure it and to remove the configuration assigned to it.
Spi_lld_send is called in order to transmit only the data over the TDI line contained into the TDI buffer, this function takes as parameter the SPI driver handler, the number of bytes to send, and a pointer of *uint8_t* type pointing to the data buffer to transmit.
Because the TMS buffer and the TDI buffer must be transmitted at the same time, in order to synchronize the JTAG state machine, this function is called by passing it as parameter the slave SPI handler, in this way the slave SPI will set its data register with data to transmit but the transmission will not start until the slave SPI will not receive the clock signal coming from the master SPI.
Spi_lld_exchange is called by passing to it the master SPI handler in order to transmit data over the TMS line, contained in the TMS buffer, and receive at the same time data from the TDO line, saved into the TMS buffer.
A call to *spi_lld_exchange* will start data transmission over both TDI and TMS lines, that because this function is called by passing the master SPI handler as parameter, so when this function is in execution, the master will output the clock signal that will drive both the DUT and the slave SPI peripheral.

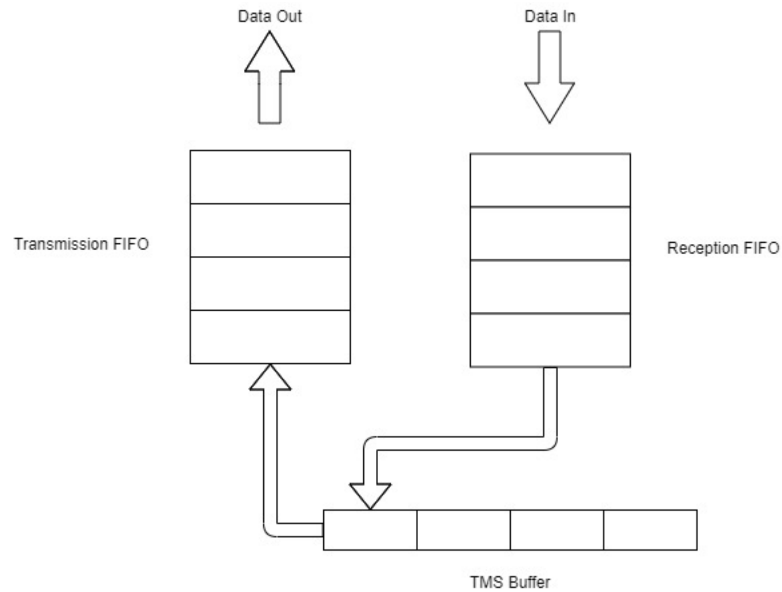

Figure 6.3: Data is sent from TMS buffer and at same time received from TDO line.

DMA and interrupts are used to manage data transmission. The DMA is used to put data to transmit from the memory into the SPI shift register and vice versa, meanwhile the interrupts are used to signal to the CPU that transmission has been completed, in this way the CPU will leave the blocking loop and continue code execution. Even if the use of DMA frees the CPU from having to manage the transmission, due to the difference between the system clock and the transmission clock it is still necessary to block the CPU, until the transmission is complete, in order to avoid the corruption of data buffers. A status flag declared in both JTAG and ATPG modules is used to check transmission status.

Chapter 7

JTAG module

7.1 Chapter introduction

The JTAG module is used to let the supervisor microcontroller to transmit JTAG data and JTAG instruction to the DUT in respect of the JTAG standard. The JTAG transmission has two main purposes: first one is to let the DUT to enter into test mode, second one is to let the supervisor to change the DUT configuration and the code saved on the DUT flash memory during the testing phase.

The purpose of this module is to allow the user to transmit JTAG data correctly managing the data shifting, checking the current state of the JTAG state machine (discussed in section 2.3) and implementing some special functions that allow the supervisor to bring the DUT in reset state or to read or write its data register.

The module is fully integrated with the FreeRTOS CLI and is built accordingly with the module architecture discussed in the previous chapter.

7.2 JTAG data type

Data buffers and status flags are defined into two separated structures, ordered (like the APIs) into layers: the lower layer data structure contains data buffers and other variables, while the highest layer contains some status flags and two different structures: the former is the low level structure, the latter is a structure containing a function pointers (discussed on paragraph 7.3).

Again, dividing the data structures into two levels ensure better code management, the user does not need to use the low level structure fields, and the data structure is declared once in the *JTAG.c* file and declared as external in the *JTAG.h* file so it can be used as soon as the module has been imported.

The low level driver JTAG structure contains:

- *hMaster* and *hSlave*: are the handlers for the SPI driver. They are pointers to the SPI drivers used during data transmission.

- *Master_spi_config* and *slave_spi_config*: other 2 handlers referring to the configuration of the SPI drivers.
- *TdiBuffer*: the buffer containing data to transmit over the TDI line.
- *TmsBuffer*: the buffer containing data to transmit over the TMS line.
- *RcvStr*: A character array that will be filled with the received data converted from *uint8_t* to character in order to be compared with other data or to be printed on the terminal.
- *Nbit*: *uint16_t* variable containing the number of bits of data to transmit. This variable is used to fill correctly the TMS buffer.
- *Byte*: *uint16_t* the variable containing number of bytes to transmit. This variable is used to set the DMA with the number of bytes to transfer.
- *Tap*: a variable used to have trace of JTAG state machine reached state.

JTAG high level data structure contains:

- *IpStat*: this variable signals the status of the JTAG driver. It can assume three different values: *JTAG_DRIVER_INIT, JTAG_DRIVER_START, JTAG_DRIVER_STOP*.
- *Lld_jtag_struct*: this is the low level JTAG structure.
- *Status*: a variable containing the error value. If some error occurs, this variable will be set with the correspondent error code.
- *OpStatus*: this variable is used to control if every called function has correctly completed its execution. If some error occurs, then the value of this variable will be *FAIL*, *DONE* on other cases.
- *Jtagop*: is the structure containing the pointers to the low level driver functions.

JTAG low level structure does not contain the TDO buffer. This choice was taken because data can be correctly received on same buffer used to transmit the TMS data.

```
//--------low level struct driver.--------
typedef struct {

    //-------SPI master and slave. Master send tms, slave send tdi. Slave work whit master's clock.-------
    SPIDriver *hMaster;
    SPIDriver *hSlave;

    SPIConfig *master_spi_config;
    SPIConfig *slave_spi_config;


    //--------JTAG buffer's definition-------
    uint8_t tdiBuffer[JTAG_BUFFER_SIZE];
    uint8_t tmsBuffer[JTAG_BUFFER_SIZE];

    char rcvStr[JTAG_BUFFER_SIZE * 2];

    //--------jtag bits & byte definition-------
    uint16_t nbit;
    uint16_t byte;

    //-------jtag flag field definition-------
    typedefTapCmd   tap;

    typedefStatus status;

}typedef_lld_jtag;
```

Figure 7.0: JTAG low level driver structure

```
//-------JTAG Driver struct-------

typedef struct {

    //-------Driver status flag-------
    jtagDriverStatus_t ipStat;

    //-------low level driver-------
    typedef_lld_jtag lld_jtagS;

    //-------flag fields-------
    typedefStatus status;

    //operation result
    opStatus opstatus;

    //-------JTAG functions pointer.-------
    jtagops_t jtagop;


}JTAGDriver;
```

Figure 7.1: JTAG driver structure

7.3 JTAGops structure

The *JTAGDriver* structure contains a field named *jtagops* of the defined type *jtagops_t*, this is a structure containing a set of pointers used to refer to the low level driver functions, and in this way the low level functions are never called directly, when necessary the pointers are used instead.
The advantage of this approach consists on the possibility for user to define its own low level functions to assign to the function pointers. If the used microcontroller belongs to the SPC5 microcontroller family, the low level driver functions are already defined.
The structure contains these pointers:

- *Lld_jtag_start*
- *Lld_jtag_stop*
- *Lld_jtag_send*

```
//-------JTAG ops struct definition-------

typedef struct {

    void (*start)(typedef_lld_jtag *jtag, const SPIConfig *spicnf1, const SPIConfig *spicnf2);
    void (*stop)(typedef_lld_jtag *jtag);

    opStatus (*send)(typedef_lld_jtag *jtag);
}jtagops_t;
```

Figure 7.2: jtagops_t structure

All of *jtagops_t* function pointers take as parameter the low level driver structure, according to the layers architecture (low level functions use low level data structure, high level functions use high level data structure).
lld_jtag_start function takes two pointers to the SPI configuration structure, one for master SPI and one for slave SPI. This function calls the low level driver SPI start function.
The *lld_jtag_stop* function takes only the low level driver structure as parameter and it is used to stop the SPI driver and JTAG driver. At last, *lld_jtag_send* function again takes as parameter the low level driver structure and it is used to

transmit data over SPI. Is possible to change the signature of the function pointers to adapt them to the driver functions of the used microcontroller.

7.4 JTAG module functions

As discussed on chapter 6, the modules are built by dividing their functions into three layers: the first layer is the wrapper layer, is composed of these functions that the user calls in order to use the JTAG module, the second layer is the transport layer, it is composed of these functions that are used in order to prepare the data to be correctly transmitted, the last layer, the low level driver layer, is composed by these functions that act as a wrapper for the peripheral low level APIs.
Starting from the upper layer, the wrapper layer, functions that compose it are:

- *JTAG_init*
- *JTAG_start*
- *JTAG_stop*
- *JTAG_transaction*
- *JTAG_read*
- *JTAG_checkRcvData*
- *JTAG_buffToString*
- *JTAG_reset*

First three functions are used in order to initialize the JTAG driver assigning to the SPI handler, contained inside the JTAG driver structure, the pointer to the SPI driver and setting all the buffers and the flags to a consistent state, to start the JTAG driver assigning the selected SPI configuration to the master SPI and the slave SPI and to stop the JTAG driver when is not anymore used by clearing the buffers and stopping both the SPI driver too.
JTAG_transaction is used to do a generic JTAG transaction, with this function is possible to set where bring the state machine before and after the data was transmitted. This is the most important function of all the module.
JTAG_checkRcvdata is used to convert data contained on receiver buffer from *uint8_t* to characters and compare it with the expected data.
JTAG_reset bring the state machine to TEST-LOGIC/RESET state.

JTAG_read transmit a specific number of bits set to 0 and it is used when the user wants to read the content of some data register.

Looking to the second layer, the transport layer, the functions that compose it are:

- *JTAG_setTDI*
- *JTAG_setTMS*
- *JTAG_send*

JTAG_setTDI is used to correctly fill the TDI buffer, contained inside the low level driver structure, accordingly to the selected SPI shift direction, same work does the *JTAG_setTMS* function but on the TMS buffer. Last function, *JTAG_send*, is used to call the JTAG low level driver function to send data. *JTAG_send* is a blocking function, so until the transmission is not complete, the CPU must stay blocked inside a loop waiting for the interrupt that signals the completion of transmission.

At last, the low level driver functions, which act as a wrapper for the SPI peripheral functions and are used to configure low level JTAG driver structure, they are:

- *Lld_jtagHwConf*
- *Lld_jtag_init*
- *Lld_jtag_start*
- *Lld_jtag_stop*
- *Lld_jtag_send*

Lld_jtag_init is used in order to initialize the JTAG driver by assigning a concrete value to the structure fields and cleaning all the buffers, *lld_jtagHwCnf* is used to assign the SPI peripheral handlers to the structure SPI handlers.

Lld_jtag_start is used in order to assign the SPI peripheral configuration and start the peripheral. This function starts the JTAG driver too, meanwhile *lld_jtag_stop* is used in order to stop both the SPI peripheral and the JTAG driver.

Lld_jtag_send is called when data is ready to be transmitted over the TMS and the TDI lines. This function calls the low level driver *spi_lld_exchange* function for transmitting the data over the TMS line with the master SPI and receive at the same time data over TDO line, putting it into the TMS buffer, meanwhile TDI data is sent from the slave SPI, without receiving anything, by calling *spi_lld_send* function.

All the low level driver functions but *lld_jtag_init* and *lld_jtagHwCnf* are declared as static functions, in this way they are not visible to the user and in addition, this functions are the default ones implemented for the SPC5 microcontroller family. If another type of microcontroller is used, the user must define its own low level driver functions that will replace the *lld_jtag_send, lld_jtag_start* and *lld_jtag_stop* functions, assign the new defined functions to the low level driver function pointers (changing the pointers signature if necessary) and call them by using the pointers.

```
//-------Low level function prototypes definition-------
static void lld_jtag_start(typedef_lld_jtag *jtag, SPIConfig mCnf, SPIConfig sCnf);
static void lld_jtag_stop(typedef_lld_jtag *jtag);
static opStatus lld_jtag_send(typedef_lld_jtag *jtag);

void lld_jtagHwConf(typedef_lld_jtag *jtag, SPIDriver *hspiMs, SPIDriver *hSpiSl);
void lld_jtag_init(typedef_lld_jtag *jtag, jtagops_t *ops);

//-------High level JTAG functions definition-------
opStatus JTAG_init(void);
opStatus JTAG_start(JTAGDriver *hjtag);
opStatus JTAG_stop(JTAGDriver *hjtag);
opStatus JTAG_send(JTAGDriver *hjtag);

opStatus JTAG_setTMS(JTAGDriver *hjtag, const uint8_t tap, typedefTapCmd nxtState);
opStatus JTAG_setTDI(JTAGDriver *hjtag, char *data);
opStatus JTAG_checkRcvData(JTAGDriver *hjtag, char *data);
opStatus JTAG_buffToString(JTAGDriver *hjtag);

opStatus JTAG_reset(JTAGDriver *hjtag);
opStatus JTAG_read(JTAGDriver *hjtag, uint16_t nBit);
opStatus JTAG_Transaction(JTAGDriver *hjtag, typedefTapCmd dst, char* data, uint16_t bits, typedefTapCmd nxtState);
```

Figure 7.3: JTAG module function prototypes

7.5 How data to transmit is managed

When data must be transmitted with the JTAG, the user must call only the *JTAG_transaction* function and pass to it the following parameters: the first and last states to reach, data to transmit and the number of bits to transmit. This function calls the other three functions coming from the transport layer: *JTAG_setTDI, JTAG_setTMS, JTAG_send* to prepare the buffers and transmit them.

7.5.1 JTAG_setTDI

This function is used to correctly fill the TDI buffer in respect to transmission ordering. Because JTAG driver is implemented to send data on LSB order, and because data that comes from CLI is wrote with MSB order and is a characters string, is mandatory, before transmit it, to convert it from characters to *uint8_t*, then reverse it in respect of LSB ordering and, at least, fill the TDI buffer.

```
opStatus JTAG_setTDI(JTAGDriver *hjtag, char *data);
```

Figure 7.4: JTAG_setTDI prototype

JTAG_setTDI works as follow:

First this function uses a pointer to the memory location given by *tdiBuffer + 1*, in this way the first location (that on array is the index zero) is not filled with data. Second passage is to check if the data to send is a 0s string, if so, the buffer will be filled with 0s and the function will return.

If not, then the string length of the data to transmit is calculated, and every character representing the data string to transmit is parsed with following algorithm:

1. Define a counter, starting from 0, to have trace of cycle repetition.
2. Take the data at the index given by *(commandStartingAddress + stringLength – cycleCounter)* value and check if it is a valid character (it must be a hexadecimal digit to be valid so it must be a character comprised between 0 and 9 or A and F, lowercase letters are allowed), if not, the function will return with error.
3. Check the cycle counter: if it is an even number means that the character taken from string should go on low nibble of the byte to transmit, so it will ORed with the TDI buffer location pointed by the pointer to the TDI buffer, else if it is an odd number then means that the value must go to the high nibble of the byte, so it will be shifted to the left and ORed with previous value.

4. The value stored on the TDI will be used as index on a reverse matrix to find the correspondent reversed value and stored into the TDI buffer at the same location.
5. If the cycle counter has the same value of the string length and the counter is an even number this means that one last nibble compose the data to send, so it will be reversed and putted into the last indexed TDI buffer location.

After this, the function returns and the TDI buffer is filled with correct data.

```
if(*cmd == '\0'){
    uint8_t i;
        for(i = 0; i <= JTAG_BUFFER_SIZE; i++){

            //fill TDI with 0's
            *(buffPtr + i) = 0x00;
        }
}else{

    offset = strlen(cmd) - 1;
    dataPtr = cmd;

    //Fill data buffer, 2 char (1 nibble) go on 1 array location on LSB order
    for(cnt = 0; cnt <= offset; cnt++){

        //Check that characters are hexadecimal digits
        if(!validChar(*(dataPtr + offset - cnt))){
            hjtag->status = ERROR_INVALID_CHAR;
            return FAIL;
        }

        auxValue = char2int(*(dataPtr + offset - cnt));

        if(!(cnt % 2)){
            *(buffPtr) |= auxValue;
        }else{
            *(buffPtr) |= (auxValue << 4) & 0xF0;
            *(buffPtr) = reverse_matrix[*(buffPtr)];
            buffPtr++;
        }
        if(cnt == offset && !(cnt % 2)){
            *(buffPtr) = reverse_matrix[*(buffPtr)];
        }
    }
}
return DONE;
```

Figure 7.5: JTAG_setTDI function

7.5.2 JTAG_setTMS

This function is used to correctly fill the TMS buffer before transmit it.
To fill this buffer, is necessary to know the first and the last state that JTAG state machine must reach and the bits number of data to transmit.
Bits sequence that shift the state machine to the first state to reach is putted into the first byte of TMS buffer. A complete transaction starting from RUN_TEST/IDLE and returning to it can be done with 8 clock pulses, so 1 byte is sufficient to do all necessary transactions.
The algorithm implemented to fill the TMS buffer works as follow:

1. First from the bits number the data bytes to transmit are calculated.
2. The function checks if there is a remainder too, if so another byte is added to total bytes number to send.
3. Next step is to check the last state to reach, if it is RUN_TEST/IDLE state then the correct set of values are inserted into the TMS location aligned with last bit of data to transmit into the TDI buffer. If instead an update state must be reached, is not important if it is UPDATE_DR or UPDATE_IR because they are symmetric (see Figure 2.2), another value is putted inside the buffer.
4. Last step is to fill first byte with the correct value to reach first state of transaction.

```
//Set pointer to index 0 of TMS buffer
buffPtr = (hjtag->lld_jtagS.tmsBuffer);

//Check now next state where state machine will be shifted
if(nxtState == RUN_TEST_IDLE){
    if(rem && rem != 1){
            byte++;
            *(buffPtr + byte) = RUN_TEST_IDLE_MID << (rem - 1);//1 SHIFT_MID RUN_TEST_IDLE_MID
            byte++;

    }else if(rem == 1){
        byte++;
        *(buffPtr + byte) = RUN_TEST_IDLE_MID;//1 RUN_TEST_IDLE_MID
        byte++;

    }else{
        *(buffPtr + byte) = 0x80;// >> rem; //8
        byte++;
        *(buffPtr + byte) = 0x01;
    }
    byte++;
    *(buffPtr + byte) = 0;//0 RUN_TEST_IDLE_TAIL
    hjtag->lld_jtagS.tap = RUN_TEST_IDLE;

}else if((nxtState == UPDATE_DR) || (nxtState == UPDATE_IR)){

    /*
     *Update state transaction must be managed case for case.
     */
    switch(rem){
    case 7: {
        byte++;
        *(buffPtr + byte) = 0x40;
        byte++;
        *(buffPtr + byte) = 0x03;
        break;
    }
    case 6: {
        byte++;
        *(buffPtr + byte) = 0xA0;
        byte++;
        *(buffPtr + byte) = 0x01;
        break;
    }
    case 0:{
        *(buffPtr + byte) = 0x80;
        byte++;
        *(buffPtr + byte) = 0x06;
        break;
    }
    default: {
        byte++;
        *(buffPtr + byte) = 0x0D << (rem - 1);
        byte++;
        break;
    }
    }
}else{
    hjtag->status = INVALID_TAP_COMMAND;

    return FAIL;
}

*(hjtag->lld_jtagS.tmsBuffer) = tap;

byte++;
```

Figure 7.6: JTAG_setTMS function

7.5.3 JTAG transaction example

JTAG_transaction is the main function to call in order to do a generic transmission of JTAG data.

The parameters of this function are:

- JTAG driver structure
- First destination to reach
- Data to send
- Bits number to send
- Next state to reach.

This function call the *JTAG_setTDI* function passing to it data to transmit and the *JTAG_setTMS* function by passing to it the states to reach and, after that all the buffers has been filled, the *JTAG_send* function is called and a transaction is done.

An example of how the *JTAG_transaction* works is shown below.

Example:

Suppose to call *JTAG_transaction* by passing these parameters:

- SHIFT_DR is the first state machine transaction and it indicate that a data is going to be transmitted.
- 0x35486 is the hexadecimal value to transmit (0x612AC0 on TDI to respect LSB ordering).
- 20 bits to send.
- RUN_TEST_IDLE is next state where state machine have to go after data has been sent.

At the end of operations on data, buffer appears like that:

	0x20			0x18	0x01	
TMSBUFFER	0010 0000	0	0	0001 1000	0000 0001	0
TDIBUFFER	0	0110 0001	0010 1010	1100 0000	0	0
		0x61	0x2A	0xC0		

Figure 7.7: TMS and TDI buffers before transmission

tmsBuffer[0]	0x20
tmsBuffer[1]	0
tmsBuffer[2]	0
tmsBuffer[3]	0x18

tdiBuffer[0]	0
tdiBuffer[1]	0x61
tdiBuffer[2]	0x2A
tdiBuffer[3]	0xC0
tdiBuffer[4]	0

Figure 7.8: Buffers before transmission on debug session

Figure 7.9: Transmitted TMS and TDI

Same approach for some special transactions, for instance the *JTAG_Reset* that will bring state machine to TEST_LOGIC/RESET state:

- TEST_LOGIC/RESET is the state to reach.
- 0x0 is data to be send (is not necessary to send some data on TDI line)
- 5 bits compose the transaction.

At the end of the operations on data, the buffer appears like the image below (only TMS reported):

Figure 7.10: TMS buffer filled to bring state machine to TEST_LOGIC/RESET state

When the *JTAG_transaction* function execution is done, it will return a flag status value meaning operation result. It can be DONE if transaction has be done without any error, FAIL on other case.
If FAIL is returned, is possible to read the status flag contained inside the high level driver structure to check the error type.

7.6 JTAG task

The JTAG task flow is mainly based on the *taskData_t* structure ID field that hold the unique value used to recognize the received command.
The values of the ID field are declared as enumeration type called *JTAGCmd_t*, ID field can be one of following values:

- JTAG_START
- JTAG_STOP
- JTAG_TRANSACTION
- JTAG_RESET
- JTAG_COMPARE
- JTAG_READ

When the JTAG task receives a notification, it will read the ID field value in order to execute the correct command by using a switch control block.
For every SPI transmission, the JTAG driver is first started and then stopped, in this way is possible for other tasks to use the same SPI peripheral without any conflict regarding driver configuration saving in this way some resources.
If the command ID is one that require to send data, like JTAG_TRANSACTION, then data and bits number to transmit are used as parameter, the former taken from *txBuffer*, meanwhile the latter is taken from bits fields of *taskData_t* structure.

When a command execution is complete, the JTAG task will notify CLI task without passing any value, and if is necessary to send on terminal the received data, CLI task will read it from *taskData_t* structure. Because this is only a read operation, is not necessary to synchronize the tasks, CLI task will be surely on blocked state waiting for notification and passing to it no value will ensure that when notified it will only resume its code flow, terminating its command function execution and returning to listen on the UART for incoming data.

If the JTAG_STOP ID is received, then the JTAG task will set to NULL its handler pointer, then it will notify again CLI task and after it will call the *vTaskDelete* function by passing NULL as parameter to let scheduler to remove and free memory occupied by the JTAG task.

Flow diagram below shows JTAG task flow.

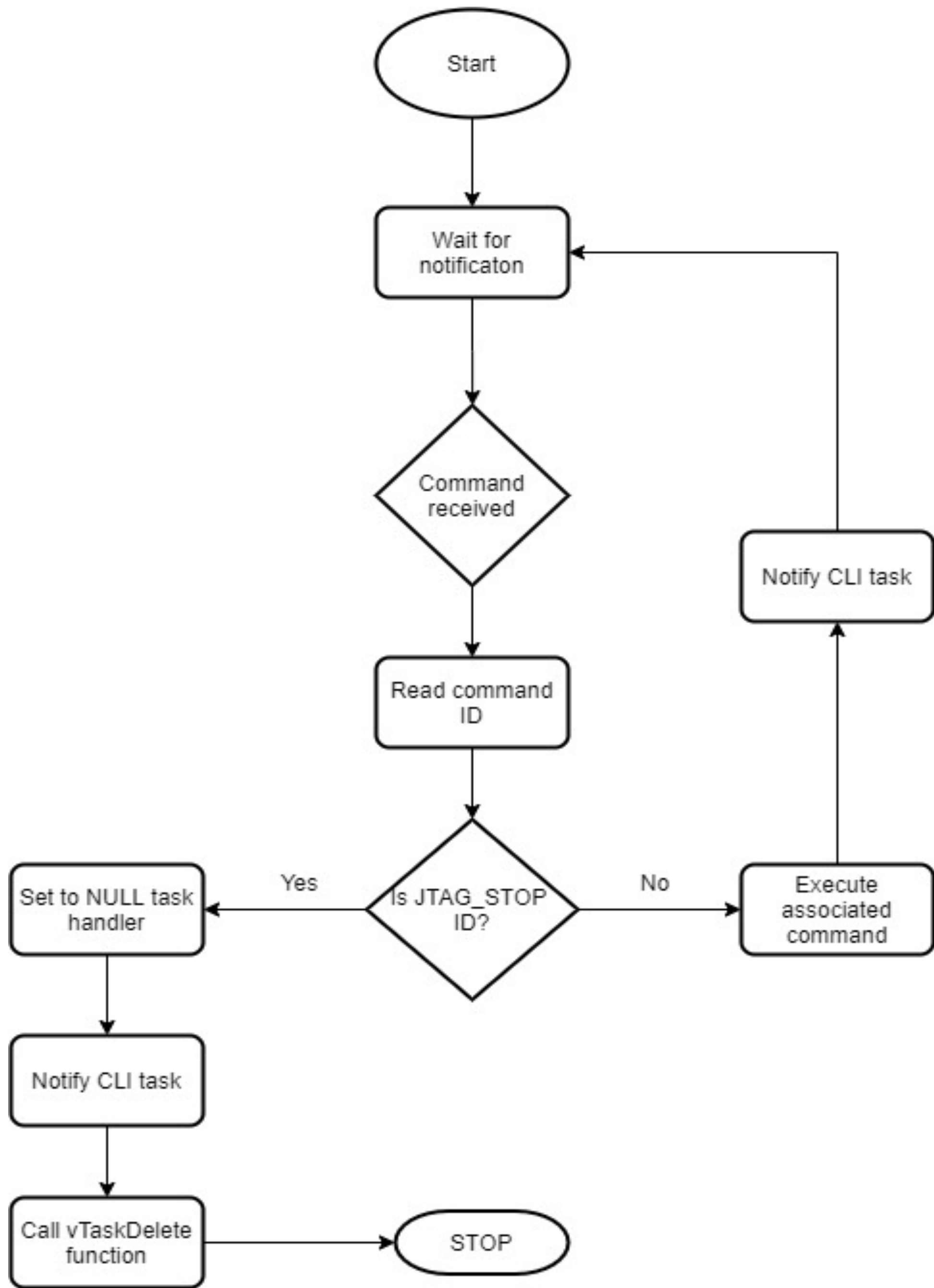

Figure 7.11: JTAG task flow

Chapter 8

ATPG module

8.1 Chapter introduction

Automatic Test Pattern Generator (ATPG) is a test strategy used to test almost all the logic that composes the tested device. During this test, some vectors or input sequences are applied to a digital circuit, and this is possible thanks to the structure of these sequences that allow the supervisor (that could be a microcontroller or a terminal) to distinguish between the correct circuit behavior and the faulty circuit behavior caused by defects.

Test patterns are not generated inside the microcontroller, they are generated outside by an external software, the ATPG module discussed here has been implemented just to provide the necessary functions to manage the patterns by moving them where is necessary.

On this chapter will be discussed how is implemented the ATPG module, explaining the module architecture, data types and functions.

8.2 ATPG data types

Like the JTAG module, the ATPG module has the same architecture so the data field and the flags are grouped into two distinct structures named *typedef_atpg_lld*, the low level driver structure, and *ATPGDriver*, the high level structure.

Fields of the low level structure are:

- *DataBuff:* a characters buffer containing sent and received data.
- *rcvStr*: a characters buffer containing received data converted from *uint8_t* to char.
- *Nbit*: variable containing bits number of the transaction.
- *Byte*: variable containing byte number of the transaction. This variable is used to set the DMA with the number of bytes to transfer.
- *CmpRes*: this variable contains the result of a compare operation. If compare is successful then the variable will hold DONE value, else FAIL.

The fields of high level structure are:

- *IpStat*: this flag is used to check the driver status. It can be: *ATPG_DRIVER_INIT, ATPG_DRIVER_START* or *ATPG_DRIVER_STOP*.
- *Atpg_lld*: this is the low level ATPG structure.
- *Atpgops*: is a structure containing the pointers to low level driver functions.
- *Status*: a variable containing the error value. If some error occurs, this variable will be set with the correspondent error code.
- *OpRe*s: this variable is used to check if every called function has been completed. If some error occurs, then value of this variable will be FAIL, DONE in other case.

8.3 ATPGOPS structure

The *ATPGDriver* structure contains, like JTAG structure, a field named *atpgops* of defined type *atpgps_t*. This is a structure that contains a set of pointers to the defined low level driver functions and they are called by using the pointers, not the function name.

The advantage of this approach consists on the possibility for user to define its own low level functions to assign to the function pointers. If the used microcontroller belongs to the SPC5 microcontroller family, is possible to use the low level driver functions already defined.

The pointers of this structure are:

- *Lld_atpg_start*
- *Lld_atpg_stop*
- *Lld_atpg_send*

Lld_atpg_start is used in order to start the peripheral used to transmit data, *lld_atpg_stop* is used to stop it and *lld_atpg_send* is used to call the implemented functions to manage data transmission.

8.4 ATPG module functions

The ATPG module functions are divided into two layers: low level functions and high level functions. First layer is composed by the functions that configure the peripheral and are used to transmit the ATPG patterns, second layer is composed by the functions callable by the user.

Low level function of the ATPG module are:

- *Lld_atpg_start*
- *Lld_atpg_stop*
- *Lld_atpg_send*
- *Lld_atpgHwConf*
- *Lld_atpg_init*

Lld_atpgHwCnf is used to initialize the peripheral and assign to it the configuration structure made by the configurator, *lld_atpg_init* is used to initialize the ATPG driver by setting it to a consistent state, and by assigning to the *atpgops* structure pointers, passed as parameter, the low level functions. If they aren't defined by the user, then the default low level functions will be used, they are discussed above.
Like the JTAG module, *Lld_atpgHwCnf* and *lld_atpg_init* are not declared as static functions, so they are visible to user.
High level functions are:

- *ATPG_init*
- *ATPG_start*
- *ATPG_stop*
- *ATPG_send*
- *ATPG_transaction*
- *ATPG_checkRcvData*
- *ATPG_buffToString*

ATPG_init, *ATPG_start* and *ATPG_ stop* are wrappers for the low level functions, so when called this functions will call the low level ATPG driver functions. *ATPG_send* call the low level send function, this is a blocking function, so task code execution will stay blocked until the transmission is not complete.
ATPG_transaction fills buffer with data to transmit, then transmit it by calling *ATPG_send* function.
ATPG_buffToString converts received uint8_t data buffer to characters string and *ATPG_checkRcvData* compares the converted data stored into the driver structure with data passed as parameter.

```
//-------ATPG low level functions definition-------

static void lld_atpg_start(typedef_atpg_lld *atpg, const SPIConfig *atpgCnf);
static void lld_atpg_stop(typedef_atpg_lld *atpg);
static opStatus lld_atpg_send(typedef_atpg_lld *atpg);
void lld_atpg_init(typedef_atpg_lld *atpg, atpgops_t *atpgops);
void lld_atpgHwConf(typedef_atpg_lld *atpg, SPIDriver *hspiMs);

//-------ATPG high level function prototypes-------

void ATPG_init(void);
opStatus ATPG_start(ATPGDriver *atpg);
opStatus ATPG_stop(ATPGDriver *atpg);
opStatus ATPG_send(ATPGDriver *atpg);
opStatus ATPG_transaction(ATPGDriver *atpg, char *data, uint16_t bits);
opStatus ATPG_checkRcvData(ATPGDriver *atpg, char* stringToCmp);
opStatus ATPG_buffToString(ATPGDriver *atpg);
```

Figure 8.0: ATPG function prototypes

8.5 Data transmission

ATPG patterns transmission is simpler to manage compared to the JTAG transmission, mainly because ATPG patterns are defined outside of the supervisor microcontroller so it must manage only the transmission process. The function that manages the transmission is named *ATPG_transaction*, it takes as parameter the ATPG driver structure, a pointer to characters pointing to the string to transmit and the bits number to transmit.

The *ATPG_transaction* function counts the bytes to transmit, then it checks the remainder of the bits number: if the remainder is not zero, another byte will be added to the total number of bytes to transmit. At the end, the low level driver function pointed by the pointer *lld_atpg_send* contained inside the *atpg_ops* structure is called. Because the patterns are modelled from an external software they are transmitted as is, without any modification or conversion.

8.6 ATPG task

In respect of the requirements of the project, the ATPG module is managed by a task. This task, to save resources, is not created at system startup, but it is created only when the command *send_pattern* is received from the microcontroller and it is destroyed when the command execution is over.

This task has the same priority level of the JTAG task and a priority level higher than the CLI task, in this way, due to the preemptive scheduling algorithm, it goes to running state after its creation and, following the code, it goes in blocked state waiting for notification from the CLI task.

The notification value sent to the ATPG task is the address of shared structure *taskData_t*.

When the ATPG task receives a notification, it reads the byte field of the *taskData_t* structure in order to know how many bytes must be transmitted.

To avoid slowing the code execution by moving a lot of bytes together, the data string to transmit is divided into chunks of size of *ATPG_BUFFER_SIZE* macro value. If the number of data bytes to transmit is greater than this value, then the ATPG task will wait for a chunk of data to transmit, calculate the remaining bytes to send, transmit the data chunk and going to the blocked state waiting for another chunk of data.

If the last data chunk is composed by more bytes than the remaining value to transmit, then the excess bytes will be discarded. To do this the task calculates for every data chunk the remaining amount of bytes to send.

If the string bytes are less than the input buffer size, then all data is sent with only one transmission. At the end of all transmissions, the task will set to NULL its handler, notify the CLI task and delete itself.

During the execution of *send_pattern* command, until all data chunks are not transmitted, the CLI will be switched to ATPG mode. In this working mode, for every input string, it will not be passed to the *FreeRTOS_CLIProcessCommand* function, but will be directly passed to the ATPG task in order to be transmitted.

```
if(byteToSend < ATPG_BUFFER_SIZE){
    xTaskNotifyWait(0x0, 0x0, (uint32_t *)&buffPtr, portMAX_DELAY);

    //send all in one transaction

    strncpy(tDataPtr->txBuffer, buffPtr, ATPG_BUFFER_SIZE);

    ATPG_start(&atpg);

    ATPG_transaction(&atpg, tDataPtr->txBuffer, byteToSend);

    ATPG_stop(&atpg);
```

Figure 8.1: Code executed for one shot transmission

```
    do{

        xTaskNotifyWait(0x0, 0x0, (uint32_t *)&buffPtr, portMAX_DELAY);

        CLEAR_BUFFER(tDataPtr->txBuffer, ATPG_BUFFER_SIZE);
        byteToSend -= strlen(buffPtr);

        if(byteToSend >= 0){


            strncpy(tDataPtr->txBuffer, buffPtr, ATPG_BUFFER_SIZE);

            ATPG_start(&atpg);

            ATPG_transaction(&atpg, tDataPtr->txBuffer, strlen(buffPtr));

            ATPG_stop(&atpg);

#if VERBOSE_MODE == 1
            printf("Remaining bytes to send: %d\r\n\r\n", byteToSend);
#endif
            //notify CLI task for more data to send
            xTaskNotify(cliTask, 0x00, eNoAction);

        }else{
            //more bytes in input than the number to send entered as parameter, take only those and discard the others
            byteToSend *= -1;

            uint8_t offset = strlen(buffPtr) - byteToSend;
            strncpy(tDataPtr->txBuffer, buffPtr, offset);

            ATPG_start(&atpg);

            ATPG_transaction(&atpg, tDataPtr->txBuffer, strlen(buffPtr));

            ATPG_stop(&atpg);

            byteToSend = 0;
```

Figure 8.2: Code executed for chunk transmissions

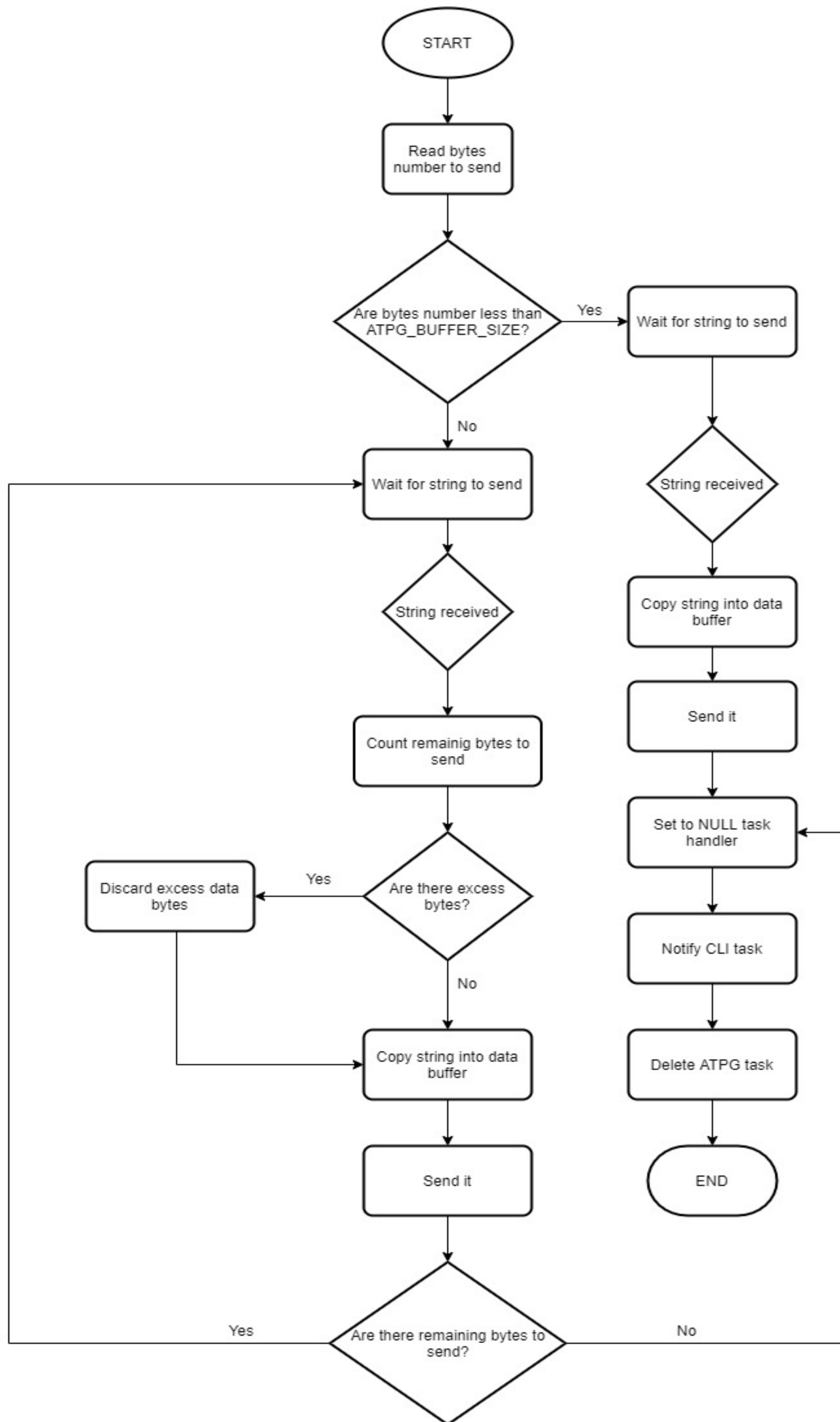

Figure 8.3: ATPG task flow chart

8.7 Sending data with GTM device

As discussed on chapter 3, the possibility of sending ATPG pattern using the GTM device was also evaluated. In this paragraph will be only discussed the possible configuration of the GTM device, in the chapter 10 will be evaluated the effectiveness of the device for this purpose.

The GTM is a device composed by a set of submodules connected each other through a routing subunit, named ARU, which main task is to route data between the submodules without any interaction whit the CPU.

This device was chosen mainly because it can automate the transmission process, meanwhile the CPU can manage the other peripherals.

The GTM used submodules are: the ATOM submodule to shift out the signals, the TIM submodule to capture incoming signals, the PSM submodule to store data coming from the CPU before being routed by the ARU submodule to the other submodules, the CMU submodule to generate the clock signal for the enabled submodules and finally the MCS that is used to manage automatically the enabled submodules.

Every submodule of the GTM-IP is composed by some channels that can work independently from each other, for the ATOM submodule two channels are used: one to shift out the clock signal, one to shift out the data, same number of channels are enabled on the PSM submodule: the former is used to store data that goes from the CPU to the GTM, the latter is used to store data that goes from the GTM to the CPU.

8.7.1 ATOM submodule configuration

As discussed on chapter 3.4.4, the used ATOM module channels are configured to work in SOMS mode, in this way, the *ATOM[x]_CHANNEL[y]* data register acts as a shift register and can be used to shift out the data.

Every ATOM channel is composed by some registers, the most important are: the CM0 register, which in SOMS mode contains data to be shifted out (each data is a word of 24 bits), the CM1 register which will contain bits number to shift out, the

SR0 and the SR1 registers that act as a shadow registers for the CM0 and the CM1 registers, the CN0 register that acts as a counter register and the *ATOM[x]_CHANNLEL[y]_CTRL* register, where is possible to configure enabled ATOM channel defining the working mode, the clock source, that in SOMS mode defines the shift speed, the shift direction and so on.

Two ATOM channels are used: the former is used to shift data that is routed from the MCS submodule to the ATOM channel by the ARU submodule, the latter is used to shift the clock pattern: it is treated as a data, with value $0x5555_{16}$ that represents the clock signal.

Every ATOM channel working on SOMS mode can manage data shifting in four different ways depending on value of three bits:

- ARU_EN: this bit defines if the ARU is enabled or not. The channel that shift data have this bit set, so the ARU will route data coming from the MCS to the channel, meanwhile the channel that is used to shift out the clock signal does not uses the ARU.
- UPEN_EN: Update Enable bit if set allows to the ARU or the CPU to update the content of the CM0 and the CM1 registers and the clock signal source for the channel. Both clock and data channels have this bit set.
- OSM: One Shot Mode (OSM) bit if set stop data shifting when there is no new data inside the CM0 or the CM1 registers. Both channels have this bit set, in this way when no more data comes from the ARU, the transmission will be stopped.[24]

The clock signal is treated as a data to shift out in order to let data channel and clock channel to start and stop the transmission at the same time. This is an important aspect because of the total number of bits to shift out is defined inside CM0 register, but the number of bits that are visible at *ATOM_OUT* is equal to CM0+1.[24]

If the clock shifting and data shifting stop at the same time, the bit CM0+1 will not be sampled.

Clock frequency is the same for the clock channel and the data channel.

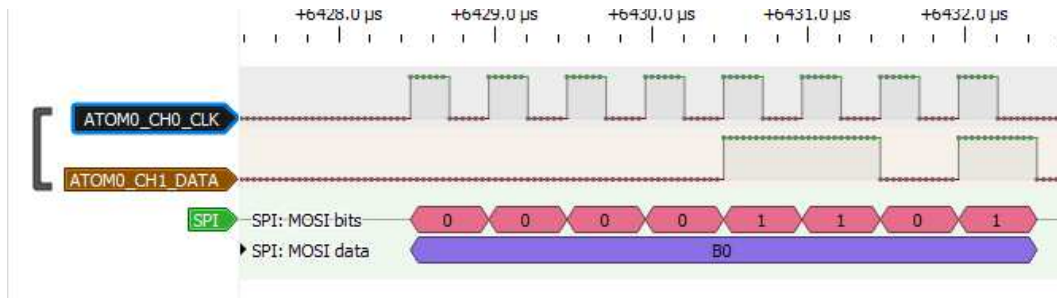

Figure 8.4: Data transmission example ($0xB0_{16}$)

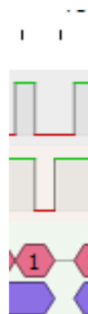

Figure 8.5: Bit CM0+1 detail.

8.7.2: PSM submodule configuration

The Parameter Storage Module (PSM) is the submodule responsible for saving somewhere data that must be moved between the GTM-IP and the CPU.

This submodule is composed by a FIFO, divided into 8 independent channels named streams, and three interfaces: the F2A that allows to the ARU to communicate with the FIFO, the AFD interface that allows the CPU to communicate with the FIFO and the FIFO itself.

On the PSM module, two streams are enabled: the first one is used to route data from the CPU to the MCS submodule (then data is moved from the MCS to the ATOM submodule), the second stream is used to store data that comes as input on the TIM submodule.

The FIFO is organized into words of 29 bits each, where 24 bits compose the data bits and the last 5 bits compose the ARU configuration bits. Because the word that the ARU can route is 53 bits wide, the ARU takes two words from the FIFO at a

time and uses them to fill the high data register and the low data register. When the data is routed to the ATOM, the low data go inside the CM0 register while the high data go inside the CM1 register.[24]

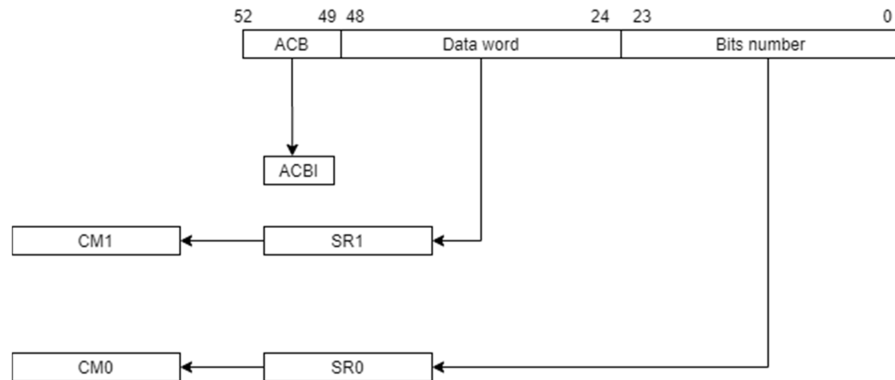

Figure 8.6: Data routing to ATOM module.

When the FIFO is filled with the data to transmit, first the number of bits to transmit is putted into the FIFO, this data will be written into the CM0 register of the ATOM data channel, then the data word to shift out is putted into the FIFO, this word will be written into the CM1 register of the ATOM data channel.

At system startup, the FIFO can be configured by setting the starting and the ending address that together define the FIFO size, and is possible to set the CPU direct access too, in this way the CPU can directly put data into the FIFO (by default, this option is disabled). Each FIFO stream can have a depth of one thousand words maximum and, when defining the starting and the ending address from each FIFO stream, is important to be careful to not overlap the two streams.

The data that comes in input of the TIM submodule is directly putted into the FIFO.

8.7.3 TIM submodule configuration

Data that comes as input into the supervisor device cannot be sampled from the ATOM submodule because it acts only as an output submodule. To let input data to be sampled, a Timer Input Module (TIM) is used.

The TIM submodule, as the ATOM submodule, can be configured to work on a specific manner. In this case, the chosen working mode is the Tim Bits Compression Mode (TBCM): in this working mode, when the TIM module receives (on the channel 0) a specific trigger, the value of the every channel of the enabled TIM submodule is sampled and every sampled value will be wrote inside the GPR1 register[24] and then putted into the FIFO.

The TIM module used to capture the input signal is set to work on TBCM mode, the trigger signal is given from the rising edge of the signal in input on the channel 0. Because the channel 0 receives as input the clock pattern shifted out from the ATOM clock channel (the one which work in SOMS mode and shifts out the clock pattern) for every rising edge of this signal, the other channels will be sampled and the composed word will be stored inside the FIFO.

Since it is not possible to operate on the data before it has arrived in the FIFO (between the TIM module and the FIFO it has been chosen not to intervene the MCS module) the CPU will have to read and empty the stream of the FIFO containing the received data every time an interrupt will be received, and compose the sequence of bits by extracting the only bit that interests (which depends by the used channel).

8.7.4 MCS submodule configuration

The MCS submodule is responsible to automate the transmission process after the data has been routed from the CPU to the GTM via the PSM submodule.

The Multi Channel Sequencer (MCS) is configured with two channels enabled: the channel zero is used to transmit the clock meanwhile the channel one is used to transmit data.

The code to be executed by the MCS module is written using a set of sub-module instructions, it is compiled at compile time together with the rest of the code for the supervisor microcontroller and it is loaded into the MCS RAM memory during the microcontroller startup phase.

After the code loading is complete, there will be a handshake between the CPU and the MCS module, in order to ensure that the module is working correctly. After this phase, the loop to transmit data is executed:

1. After the handshake phase, both channel 0 and channel 1 are disabled waiting for a trigger.
2. When some data must be transmitted, the CPU write inside the R1 register of the channel 1 the number of bytes to transmit and set the bit 1 (by writing the value 0x02) inside the STRG (Set Trigger Register) in order to trigger the channel 1.
3. When channel 1 receives the trigger, for every loop cycle it reads two words from the FIFO using the ARU, the first word read is the number of bits to shift out, this value will be written into the R2 register, the second word represents the bytes to transmit and it will be written into the R3 register. After this write phase, the channel 1 will first use the instruction to write data into the ARU data registers and let it to pass the data to the ATOM data channel, then it will write the value 0x01 into the STRG register triggering the channel 0.
4. At this point the channel 0 will use the ARU to route the clock pattern and the bits number which compose the clock pattern (16) to the ATOM channel 0 SR0 and SR1 registers. The ATOM channels are enabled to shift, so after the values were written inside the registers, the transmission will begin. The

used GTM clock speed allows to put data into the ATOM registers before the transmission starts.

5. At last the channel 0 will be disabled and it will wait for a trigger from the channel 1, meanwhile the channel 1 will decrement the loop counter and, if it is not equal to zero, repeat the cycle again.

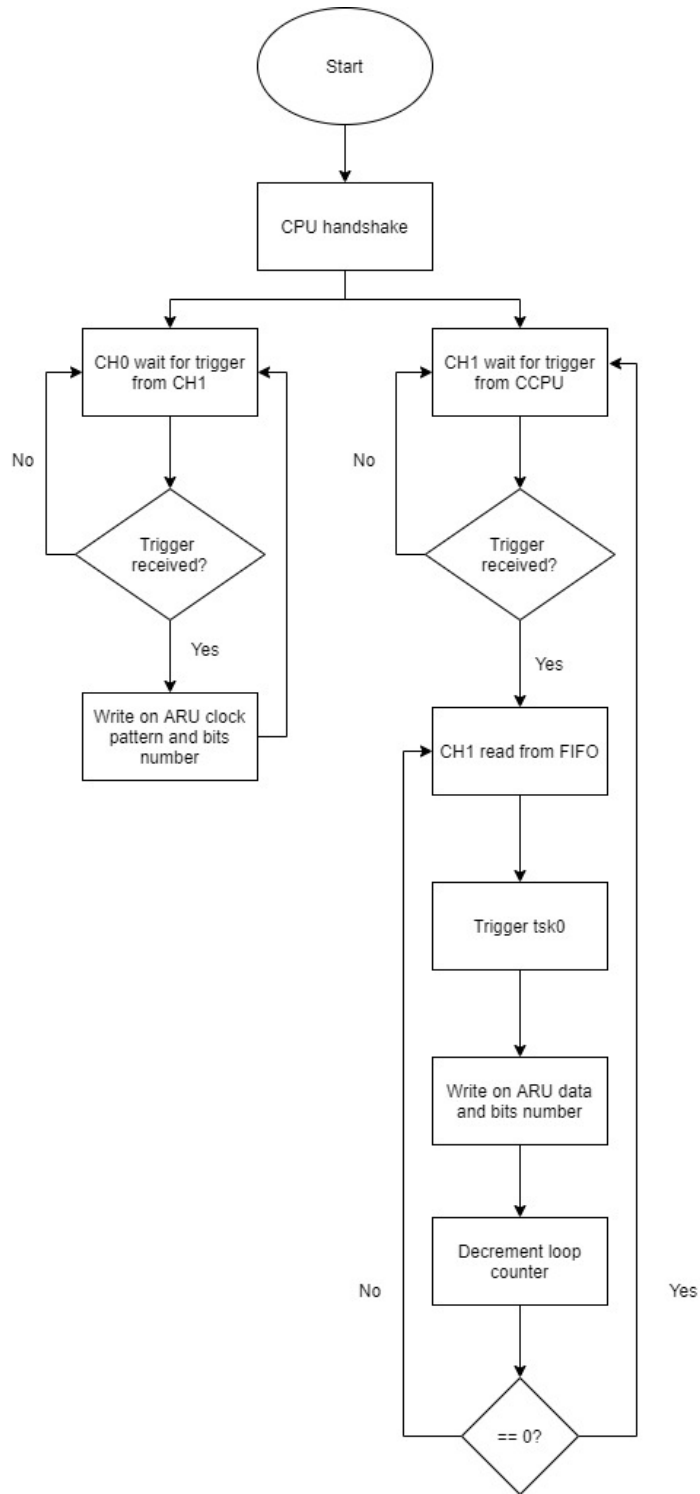

Figure 8.7: MCS code flow.

```
;---------------------------------
;Task 0 manages clock transmission
;---------------------------------
tsk1_init:
        movl R7 (tsk0_stack-4)                  ; initialise stack pointer of task 1
        movl R0, 0
        movl R1, 0
        movl R2, 0
        movl R3, 0
        movl R4, 0
        movl R5, 0
        movl R6, 0


;----------------------------------------
;Prepare registers for clock transmission
;----------------------------------------

        orl     R0      0x000002
        orl     R1      0x000010                ;load clock pattern into R5 register
        orl     R2,     0x005555                ;load bits number to shift out
LOOP:   wurm    R0,     STRG,   0x000002        ;task0 must wait until it is triggererd by task1

;trigger received
;----------------
        orl     CTRG,   0x000002                ;clear received trigger
        awr     R1,     R2,     ARU_PORT1       ;write clock pattern to ARU
        jmp     LOOP                            ;jump to tranmsission loop waiting for the other trigger



;----------------------------------------------------
;Handshake between tsk0 and CPU
;----------------------------------------------------
MCS_HANDSHAKE:

        orl     STRG,   0x000001
        orl     R0,     0x000001
        wurm    R0,     STRG            0x0001
        mov     R1,     TBU_TS0
        orl     R0,     0x0061A8
        add     R0,     R1
        wurm    R0,     TBU_TS0,        0xFFFF
        orl     CTRG,   0x000001

;here goes loop to manage multiple transmissions
;----------------------------------------------

        movl    R6,     0x051                   ;FIFO address where read from
        movl    R1,     0x000001                ;number of bytes to transfer

;here data is transmitted
;-----------------------
TX_LOOP:
        ardi    R2,     R3                      ;read data from ARU and put it on R2 and R3 registers. R2 hold bits number, R3 holds data to transmit
        call    TX_DATA                         ;call transmission routineD
        subl    R1,     0x000001                ;decrement loop counter
        jbc     STA,    Z,      TX_LOOP         ;if R1 is zero, then all bytes were transmitted
        ret



;-----------------------------------------
;data transmission routine executed by tsk0
;-----------------------------------------

TX_DATA:

        awr     R2,     R3,     ARU_PORT0       ;do a blocking write to ARU. R2 holds lower 24 bits, R3 upper 24 bits, ARU_PORT1 is the ARU pool where to write
                                                ;after this instruction data should be transmitted
        orl     STRG,   0x000002                ;trigger task0 to transmit clock signal by writing on STRG register
        ret                                     ;return from transmission routine
```

Figure 8.8: MCS assembly code for channel 0 and channel 1.

Chapter 9

Guided User Interface (GUI)

9.1 Chapter introduction

During a test session a lot of commands could be transmitted from the terminal to the supervisor microcontroller. To improve the efficiency of the tests an aspect that can be addressed could be the automation of the transmission process of the commands.

To reach this scope, a Guided User Interface (GUI) was implemented. The GUI allows to send the commands implemented for the JTAG and ATPG modules by writing them on a file with extension *.txt*, upload it on the GUI and then let the GUI read the file and send it one row a time to the supervisor. The advantages in the use of the GUI are all related to the time taken from the user to insert manually all the commands. One example could be the case of the JTAG instruction to be sent every time user want to put the DUT in test mode, in this case is necessary to send a lot of instructions and the use of a system that allows to group all instruction inside a single file and let to send it automatically is a great improvement. This chapter gives an idea of a possible GUI to develop in order to achieve the above discussed result. The GUI is not used in this thesis.

9.2 GUI architecture

The code of the GUI is written using Python, a high level, interpreted, multi-purpose programming language that allows to efficiently build the GUI thanks to the big amount of free libraries.

Python is an object-oriented programming language, so is possible to define classes and create objects that simplify the code management and the integration with other modules. Python version used to write the GUI is the v3.8, to implement the GUI TKinter module is used in order to manage the graphical aspect of the GUI, while the pySerial module, downloadable from the python official site

https://pypi.org/project/pyserial/, is used to manage the serial connection with the supervisor microcontroller.

The GUI basically consists of two classes: the former is a wrapper for the pySerial module, in order to manage the serial connection and control the correctness of the settings before trying to establish a connection with the microcontroller, this class is called *jSerial*, the latter is the one that reads the file, passed by path from the GUI, check row by row if the written commands are correct and then pass it to an object of the *jSerial* class in order to transmit every row through serial connection to the microcontroller, this class is named *jFile*.

The terminal waits for an acknowledgement from the microcontroller in order to know when the command has been executed, so the code is written without the use of the tasks.

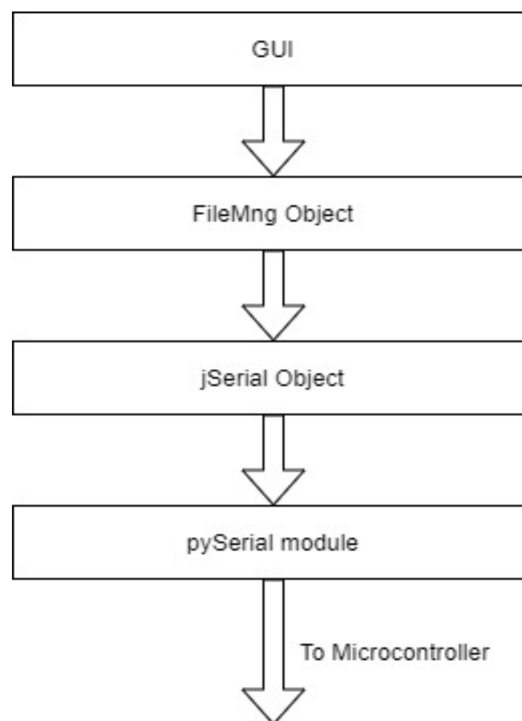

Figure 9.0: GUI architecture

9.3 jFile class

The *jFile* class was implemented in order to simplify the file read and commands check.

This class contains the necessary methods and the data structures in order to read a file, check if the file is well-formed, and add at the end of the command string a carriage return and line feed characters *\r\n* in order to let the microcontroller recognize where the command string ends.

If during the file read a command is not written correctly, then the read process will be stopped and an error will be reported. Checking the file correctness before sending it to the supervisor microcontroller allows to reduce time avoiding the exchange of error messages between the microcontroller and the terminal.

9.3.1 jFile class attributes

The attributes of this class are:

- *CmdFile*
- *cmdList*
- *atpg_mode*
- *error*
- *commands*

CmdFile is a pointer to the file to read, *cmdList* is a list that will contain the read commands string, composed by command name, parameters list and a carriage return and line feed characters.

Atpg_mode is a flag that is true when the *send_pattern* command is read from the file: when this flag is true then no syntax check will be done on the file rows until all necessary bytes, specified by the parameter of the *send_pattern* command, have been read by the GUI.

Error is a variable that contains the line number where the error was found.

Commands is a string list containing the accepted commands, it is used to compare first word of every command string, with the elements of this list. If a command is not present inside the list, then an error will be reported. The comparison is not done when *atpg_mode* flag is true.

9.3.2: jFile class methods

The implemented methods of this class are used mainly to read the file passed as input and check the commands correctness.
Every line of the file is stored inside the *cmdList* list and a sequence of carriage return and line feed characters is added at the end of the command string.
Methods of the *jFile* class are:

- *__init__*
- *readFile*
- *checkCommands*
- *getErrorLine*
- *getCmdListSize*
- *getCurrLine*
- *reverseCmdList*

__init__ is the class constructor, it takes as parameter a string representing the path of the file to open, then the file is opened and the reference is assigned to the class attribute *cmdFille*, if the file does not exist an exception is raised. *ReadFile* is called to open and read the file, this method reads a command line at a time and put it into the *cmdList* then the command list is compared with the one containing the recognized commands to check correctness inside the method *checkErrors*, if no errors are reported then carriage return and line feed characters are added at the end of the string and True value is returned to signal correct execution. If some error occurs, False is returned instead.
Other methods like *getErrorLine* are getter methods used to acquire information about the GUI status, at last *reverseCmdList* method is used to reverse the content

of the command list (because every string of the list is sent starting from the last element, so it must be reversed).

9.4: jSerial class

JSerial class is a class that act as a wrapper for the pySerial module. This class contains attributes and methods used to let the terminal to connect to supervisor microcontroller trough serial connection (the UART peripheral).
The constructor of this class is called after the *Connect* button has been pressed, then the settings of connection such baud rate, parity bit and so on are taken from the GUI setting fields. If some parameter is not set, then the GUI will not establish the connection with the supervisor and an error will be reported.

9.4.1 jSerial attributes

Attributes of *jSerial* class are:

- *ser*
- *SerialDict*
- *Status*
- *ConnectionFlag*
-

Ser is a reference to the object of the class Serial (contained inside the pySerial module) created after that the connection has been established, *serialDict* is a dictionary used to map the GUI settings with the macro values of the serial class, status is a flag reporting the connection status and *connectionFlag* is a flag that is true if the GUI is connected with the microcontroller, false otherwise.

9.4.2: jSerial methods

The implemented methods of *jSerial* class are used to connect the terminal with the microcontroller, to send a row of the commands list and to listen to serial waiting for a message from the microcontroller.
The methods of this class are:

- *__init__*
- *Write*
- *Read*
- *Close*
- *getStatusStr*
- *getConnectionStatus*

__init__ is the class constructor, it takes as parameter the connection settings and pass them to the constructor of the Serial class. When the object is created, then the connection will be established too, if some error occurs, an exception will be raised. If the connection has been established then a message will be shown and *connectionFlag* will be set to true.
Write and *read* are methods used to send an entire row to the microcontroller and to listen on serial port waiting for a string from the microcontroller. Both write and read operations are blocking methods with a timeout, set to 1 second, to avoid a system block if something goes wrong, the *close* method is called when the disconnect button has been pressed, this method will close serial connection and set to false *connectionFlag* attribute.
The other methods are getter methods used to acquire information about the connection status to print on the GUI.

```python
def __init__(self, port, baud, bytesize, parity, stop, xonxoff, rtscts,
             dsrdtr):


    try:
        self.ser = serial.Serial(port, baud, self.serialDict[bytesize], self.serialDict[parity], self.serialDict[stop], None, False,
                                 False, False, True)

        self.status = "Connected to " + port + "\r\n"
        self.connectionFlag = True

    except serial.SerialException:

        #this message will be printed on window of GUI
        self.status = "Unable to connect whit target MCU"
        self.connectionFlag = False
```

Figure 9.1: jSerial class constructor.

9.5 The interface

The GUI is composed by only one window divided into four panels.

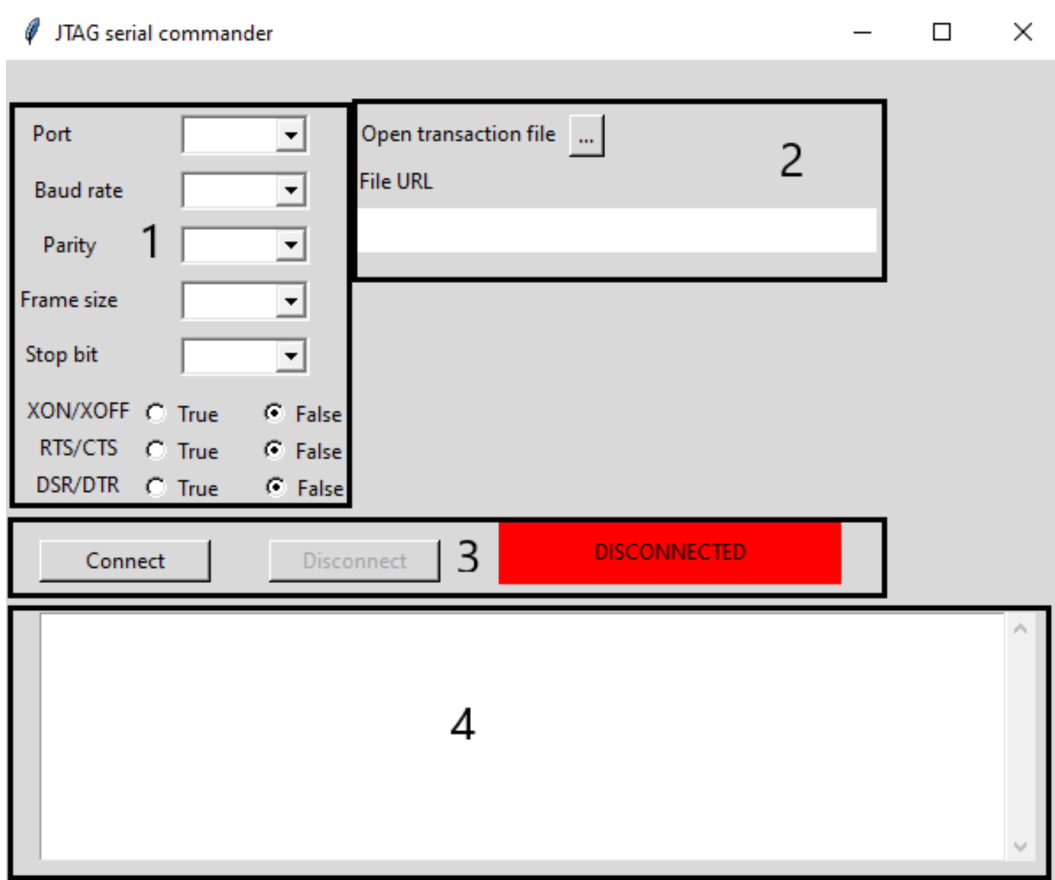

Figure 9.2: The GUI.

1. This panel lets the user to configure the connection parameters such as baud rate, parity bit, frame size and so on. The connection cannot be established

if all the parameters are not configured. After the GUI is connected to the microcontroller, this panel will become disabled so the parameters cannot be changed until the current connection is not closed.

2. This panel allows the user to choose the file to read. The file read process and error checking process start after that the file has been chosen. Is possible to choose a file even after that the GUI has established the connection with the microcontroller.
3. This panel is used to connect and disconnect the GUI from the microcontroller. The connection cannot be established if all parameters are not set on panel 1. When the GUI is connected to the microcontroller, the *Connect* button will be disabled, the *Disconnect* button will be enabled and the red icon will become green. A message will report that the connection has been established.
4. This panel shows status messages and data sent from the microcontroller.

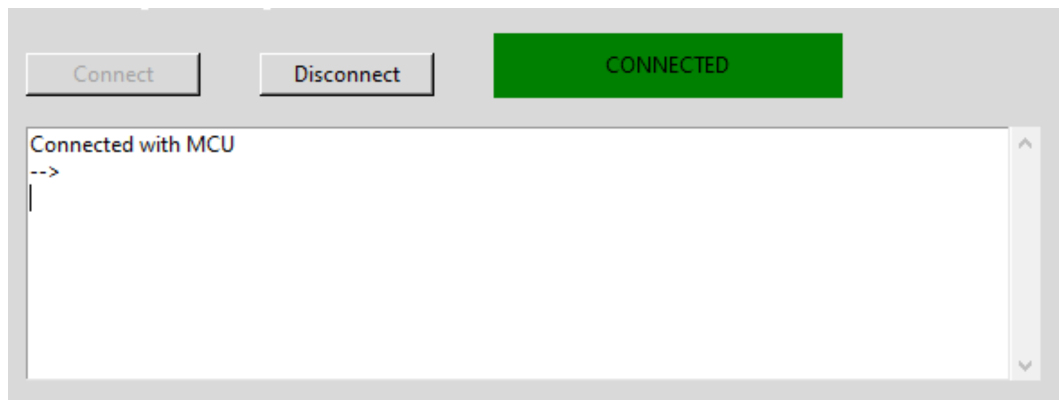

Figure 9.3: GUI connected with the microcontroller.

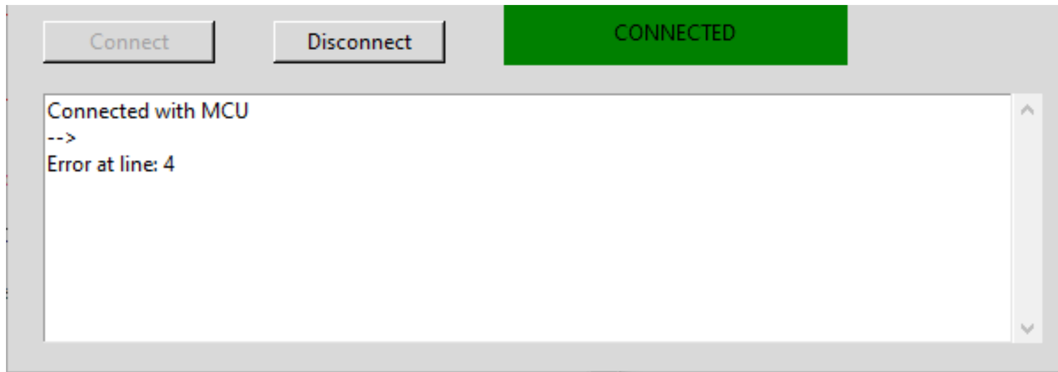

Figure 9.4: Error on file read.

Chapter 10

Conclusions

The purpose of this thesis was to discuss all the steps followed during the implementation process of the JTAG and ATPG modules, ranging from the theoretical aspects upon the testing techniques to the final software implementation. The order of the chapters is not random, the purpose of this ordering was to relate the reader to the logical process that allow the engineers to find and develop new solutions first looking to the working environment, then choosing the best suitable tools in order to create something that can be adaptable, simple and efficient.
Following this philosophy the first chapter discussed about testing and its characteristics clarifying the boundary conditions: the importance of testing on the devices, on the chapter 1, then the "protagonists" of this thesis were introduced: the JTAG standard and the ATPG methodology that were discussed in detail in chapter 2, then from chapter 3 to chapter 9 was covered all the aspects of the implementation process.
Some aspects were discussed regarding the hardware architecture of the JTAG and the ATPG, in this way was possible to choose the best peripheral to use (the SPI) and evaluate the possibility to use another peripheral: the GTM, which the possibility of use will be discussed on following paragraphs.
Next step was to evaluate the software architecture of the modules, following the same logical thread: evaluate the best tools, like RTOS, then choose the best architecture for the modules, then implementing them without never forget the three keywords of the code: structure, security, portability.
Next paragraphs will close the circle, giving an overview of the resources consumption of the implemented modules (without considering the RTOS which is not an integral part of the modules), establishing whether GTM is a valid alternative to the SPI peripheral for the moving of the ATPG patterns and, finally, showing the operation of the implemented modules.

10.1 Required resources

The required memory is divided between the memory used to store the code of the modules and the required memory used to store the code to handle the peripherals. If FreeRTOS is used too, target microcontroller must have at least 150 KB of available memory.

The tables below show the required resources for both the compiled and not compiled files and the required hardware peripheral that must and can be used.

Component	Not compiled	Compiled
SPI	122KB	45KB
UART	119KB	1KB
ATPG	9KB	1KB
JTAG	17KB	2KB
Total	148KB	51KB

Table 10.1: Amount of required resources.

Peripheral	N°	REQUIRED
SPI	2	YES
UART	1	NO
DMA	1	NO

Table 10.2: Required hardware peripherals.

How tables show, a very small amount of memory resources and a small number of peripherals are required, in this way the purpose of creating a subsystem that is efficient and does not require too much resources is reached, with a total memory requirement, for the compiled files, less than 55KB.

The small amount of required resources allows the use of the modules on a wide range of microcontrollers with only the restriction regarding the number of mounted SPI (at least two) but is possible to implement the transmission process using the GPIO (more complex to implement and manage and not discussed on this thesis) but allowing, in this way, to use the modules on all the microcontrollers.

10.2 The usage of the GTM peripheral.

As mentioned on previous chapters (chapter 3 and chapter 8), the usage of the GTM peripheral was considered. Before to define if this peripheral was a good choice for the ATPG module, was mandatory to study its functioning and its components and after doing some experiments in order to understand his strength and his weakness. Regarding the experiments, they were aimed to establish if was possible to correctly emulate an SPI peripheral.
The experiments arise two problems regarding the transmission and the reception processes: first one is the management of the synchrony of the data signal and clock signal, the second problem regards the managing of input data.

10.2.1 Synchronization of the signals

The first experiment regards the management of transmission process and it was divided into two phases: first phase is to find the best configuration to manage the transmission process, the second phase was to ensure the correctness of the emulation of the SPI.
As discussed in chapter 8, the chosen configuration for the output management involves the use of two ATOM channels (for data signal and clock signal) and the use of the MCS to automate the transmission process.
The second phase of the experiment brought to light a problem inherent in the synchronicity of the data signal and clock signal: the SPI protocol provides that, in order for the data to be sampled correctly from the circuitry, the data signal must be on a stable state at the time when it is to be sampled (depending on the SPI configuration this can be the rise or the falling edge of the clock signal). As for the GTM, since it is emulating an SPI, there were problems related to the correct synchronization of signals.

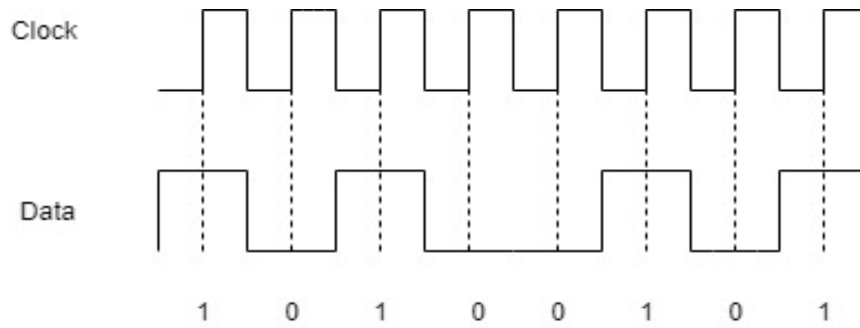

Figure 10.0: SPI clock signal and data signal synchronization

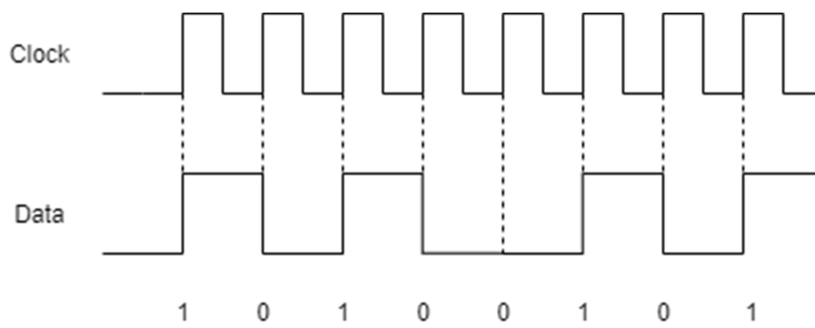

Figure 10.1: GTM clock signal and data signal synchronization

Considering the images above, first one shows the synchronism of SPI device: the data signal is already stable when the sampling edge (in this case the rising edge) of the clock signal comes, while the second image, which represents the synchronism of the signals of the GTM device, shows that the clock signal and the data signal raise together, this may lead to sampling errors as the image below shows.

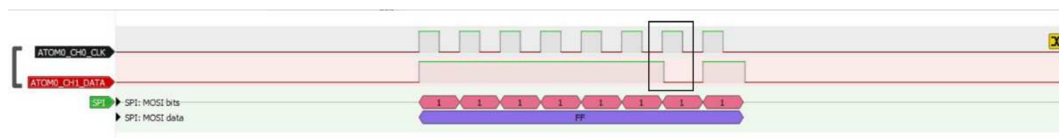

Figure 10.2: Sampling error ($0xBF_{16}$)

The sampling problem can be solved by adopting the following approach: if the clock line and the data line transmit at the same frequency, is possible to double the data bits so, for instance: if the sequence 01101_2 must be transmitted, in order

to ensure correct sampling the sequence bits can be doubled, so the sequence will become: 0011110011_2, in addition, to ensure correct sampling, the clock pattern shifted out from the ATOM clock channel must become $0xAAAA_{16}$.
Following this approach, the data signal will be ever on a stable state when the sampling edge of the clock signal comes.

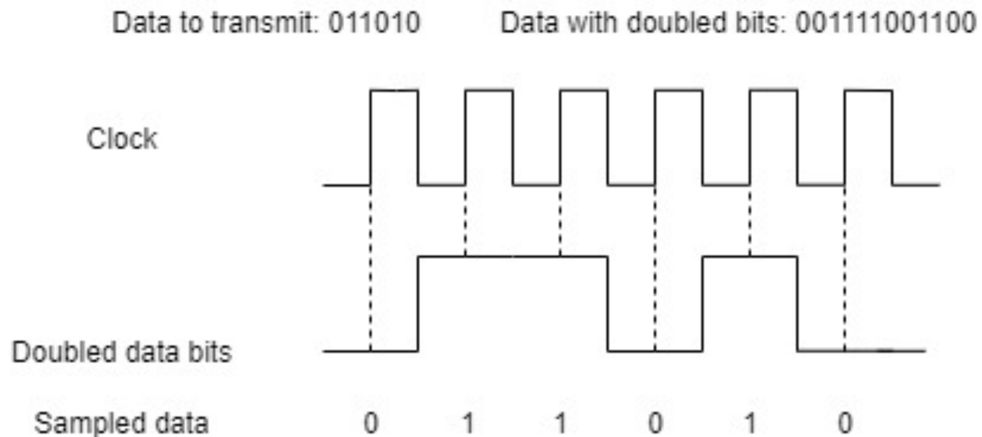

Figure10.3: Doubled data bits transmission example.

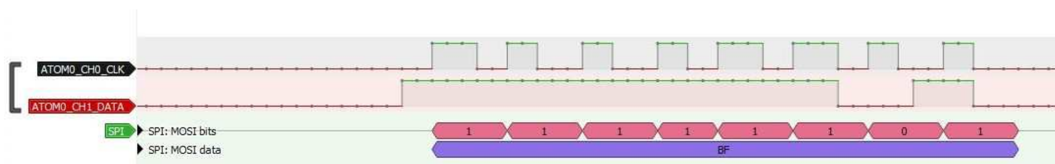

Figure 10.4: Correct sampling ($0xBF_{16}$)

But this solution is not free from problems, first one regards the bits doubling process: the software that creates the ATPG pattern is not prepared to double the bits, so should be the microcontroller to deal with the bits doubling thus increasing the effort of the CPU, second problem: the ATPG patterns are often long sequences of bits (in the order of hundred bytes), doubling the bits that make up the ATPG patterns would require twice as much memory to store the patterns with the bits not doubled.

10.2.2 Management of input signals

The second experiment was also divided into two phases in order to find the best configuration of the GTM peripheral to manage the input and to ensure that the configuration works correctly.

To ensure that the input was sampled correctly, an interrupt regarding the bit overflow (a situation where a bit is lost because the transmission happen too fast with respect to the input sampling time) was enabled on the used TIM channel.

The result of the experiment shown that, under a transmission frequency of 2 MHz, the sampling is done correctly, while with a transmission frequency greater than this value, the overflow happens. So, to ensure the correctness of transmission, the used frequency for the sampling must be less than 2MHz, but this value may be insufficient to correctly stimulate the DUT.

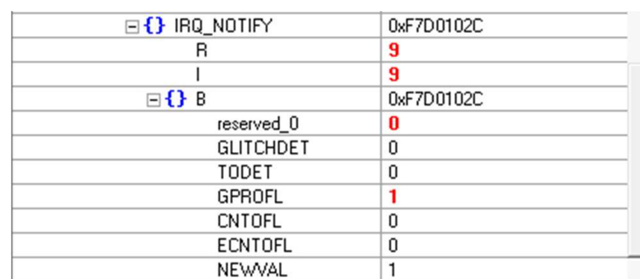

Figure 10.5: Overflow bit set.

The image above shows the overflow bit, named GPROFL, equal to one meaning that an overflow occurred.

10.2.3 Results

Confirming the results achieved, the Bosch guide that explains how to use the GTM peripheral to emulate an SPI[30] says that the GTM can emulate the SPI but is mandatory to respect a constraint in order to ensure that the transmission will occur without any error: this constraint regards the maximum allowed frequency. It can be calculated by the following inequality:

$$T > 3 * ARU_CYCLE[30]$$

Where:

- T: is the timing.
- ARU_cycle: is defined as the minimum time required from the ARU to serve one GTM submodule.

The value of ARU_cycle can be calculated as:

$$ARU_Cycle = C * GTM_SYS_CLK[24]$$

Where:

- C: is a constant depending to the used GTM peripheral.
- SYS_CLK: is the working clock frequency of the GTM device. For the experiments done this value was equal to 80 MHz.

With this SYS_CLK frequency, the value of ARU_Cycle is: 0.6l125*10^(-6), which corresponds to 0.6125 microseconds, so the value of T is equal to: 1.8375*10^(-6), or 1.8375 microseconds, that corresponds to a frequency about of 550 KHz for the used device. The transmission frequency to use to ensure the correctness of the transmission process is too low to correctly stimulate the DUT. Following the experiments carried out, the conclusion reached is that the GTM is not suitable for the transmission of ATPG patterns, a conclusion dictated both by the results achieved and by the nature of the device, thought to be used to control the mechanics of the cars, where is not necessary for the device to be able to transmit a lot of bytes with a high transmission frequency.

10.3 Modules operation

This paragraph shows how the implemented modules work. This paragraph will show the flow of operations to do starting from the transmission of the ATPG patterns from the terminal to the microcontroller, followed by the read of the DUT ID code and an example of generic transmission done in order to let the DUT enter in test mode.

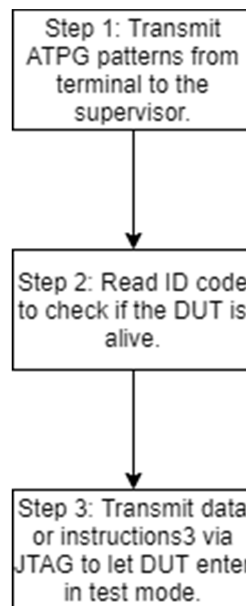

Figure 10.6: Working flow.

Starting from the beginning, once the code is loaded inside the microcontroller, if everything works and the microcontroller is able to communicate with the terminal, a prompt "→" will be displayed on it. The presence of this symbol indicates that the task that manages the CLI is listening on the UART waiting for a command.

To know the implemented commands and their syntax, the command *help* can be typed. The response of the microcontroller will be the commands list with their explanation.

```
--->help

help:
 Lists all the registered commands

jtag_transaction -- do a standard JTAG transaction
 Usage: jtag_transaction <dest> <data> <nbit> <nextState>

jtag_reset  -- send reset command to TAP
 Usage: jtag_reset

jtag_compare -- compare received data of last transaction with data passed as parameter.
 Usage: jtag_compare <data>

jtag_read -- send n bits to 0 to read content of a register.
This command do a  transaction to data register.
Usage: jtag_read <bits>

enable_jtag -- Start JTAG driver before use
Usage: enable_jtag

disable_jtag -- Stop JTAG driver after use
Usage: disable_jtag

send_pattern -- send pattern over ATPG driver
when used, this command switch CLI to ATPG mode, so data will be directly send to ATPG task
Usage: send_patter <bytesN>
```

Figure 10.7: List of available commands.

First, if necessary, some ATPG patterns can be transmitted from the terminal to the supervisor microcontroller (what is done with them depends from the test phase, they may be transmitted to the DUT or stored inside an external flash memory), this is possible by using the command *send_pattern* followed by the byte number to transmit, the command syntax is:

send_pattern <bytes_to_transmit>

As already discussed in chapter 8, if byte number to transmit is greater than the input buffer size the data will be divided into packets. When the data to transmit is composed by less than this value, the transmission will be done on a single shot. Suppose, for instance to have to send 32 bytes on ATPG, the command syntax will be:

send_pattern 32

After the command has been received, the supervisor microcontroller goes on ATPG console mode and the input data strings will be directly passed to the ATPG task without being passed to the *FreeRTOS_CLIProcessCommand*, and it will be copied inside the transmission buffer and directly transmitted.

```
Simulated I/O - connected via \\.\COM4

--->send_pattern 32
ATPG driver Started
> |
```

Figure 10.8: ATPG task waiting for the data to transmit.

The switch of the console from standard mode to ATPG mode is confirmed by the change of the prompt symbol printed in the terminal: "→" when the console works in standard mode, ">" when the console is in ATPG mode.

As mentioned less bytes than the maximum input buffer size will be transmitted on a single shot, on other cases the CLI will take all the input bytes and pass to ATPG task, then the ATPG task will calculate the remaining bytes and let the CLI task to print the value on the terminal.

When all the bytes has been sent, then the ATPG task will delete itself to free the microcontroller memory and the CLI will switch again in standard mode.

```
Simulated I/O - connected via \\.\COM4

--->send_pattern 120
ATPG driver Started
Remaining bytes to send: 120

ATPG driver Started
> iehheio3788475u45j4oi4j4oi45948ujdiojdkje89894j4io5ij4ioj5j4jl5n
Remaining bytes to send: 52

> ijljrlehjr834787954jiiodmllkedmlemk4875854ujroi4rokrkjaashshssjs
--->
```

Figure 10.9: More than one data packet transmitted.

Watches 1

Name	Value
txBuffer[40]	'5'
txBuffer[41]	'4'
txBuffer[42]	'u'
txBuffer[43]	'i'
txBuffer[44]	'r'
txBuffer[45]	'o'
txBuffer[46]	'i'
txBuffer[47]	'4'
txBuffer[48]	'r'
txBuffer[49]	'o'
txBuffer[50]	'k'
txBuffer[51]	'r'
txBuffer[52]	'k'
txBuffer[53]	'
txBuffer[54]	'
txBuffer[55]	'

Figure 10.10: Last data packet truncated.

If more bytes than the expected will be passed in input, then the ATPG task will truncate the excess bytes. Is not possible to show here the capture of the transmission of the ATPG pattern due to its dimension.

After the patterns moving is complete, the next step should be to enable the use of the JTAG with the command *enable_jtag*. If the command execution is done and the JTAG task has been correctly created, then a confirmation message will be displayed on the terminal.

```
--->enable_jtag
JTAG driver Started
--->
```

Figure 10.11: Response from microcontroller.

After the message has been displayed and the CLI task is waiting for a command, is possible to start the JTAG transmissions. First the ID code of the DUT connected to the supervisor should be read, in this way is possible to ensure that all the connections have been done in the correct way and that the DUT is alive. To read the content of a register, in this case is the ID register, depending on the device, is possible to directly read it by using the command *jtag_read* followed by the bits number to read or, if the device wants a specific instruction before to transmit the ID, a transmission with the specific instruction must be done. For the used device, the ID can be read directly with *jtag_read* command.

The tested device has ID code equal to: 0x1110041, composed by 32 bits.

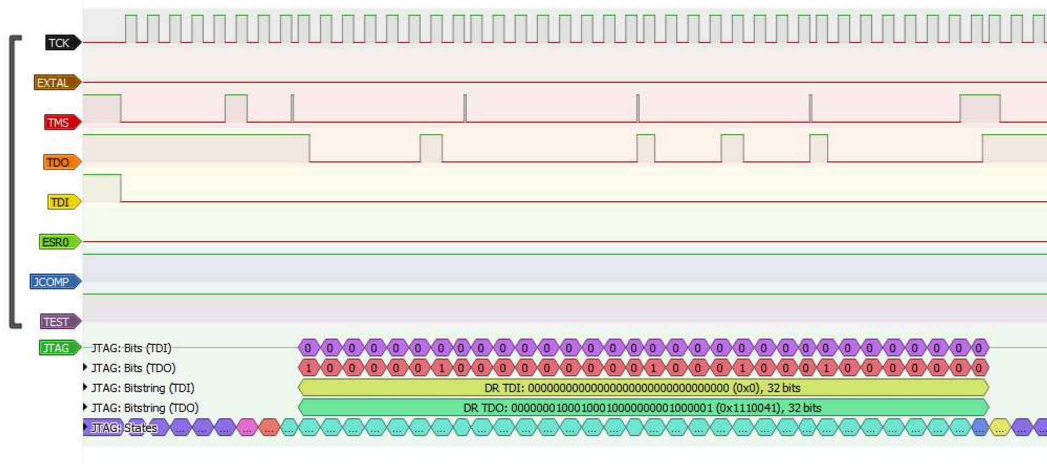

Figure 10.12: jtag_read command transmission and response.

```
--->enable_jtag
JTAG driver Started
--->jtag_read 32
JTAG driver Started
Data read: 01110041
--->
```

Figure 10.13: read ID printed on terminal.

If the read ID is correct, the user will be assured that all connections are correct and communication can take place correctly. After all the necessary verification, it is possible to transmit the instructions or data through JTAG.

To enter in test mode, is mandatory to transmit a correct sequence of data and instructions, for instance below is shown an instruction that compose the sequence. First the correct command must be called, it is *jtag_transaction* with following syntax:

jtag_transaction <first_state_to_reach> <data> <bits> <last_state_to_reach>

So the entire command will be:

jtag_transaction shift_ir 3E 6 run_test_idle

Where:

- Shift_ir is the first state to reach, this because we are going to transmit an instruction.

- 3E: is the hexadecimal value to transmit.
- 6: is the bits number that compose the instruction.
- run_test_idle: is the last state to reach.

After the command has been sent to the microcontroller, it will be parsed and the JTAG task will call the JTAG module function *JTAG_transaction* that, discussed on chapter 7, will call the transport layer functions to correctly fill the buffers followed by the call to the low level driver function to transmit them.
The image below shows what the *tDataStruct* (the structure shared by the tasks) contains before the call of the *JTAG_transaction* function.

Watches 1

Name	Value
tDataStruct	0x400694A8
ID	JTAG_TRANSACTION
bits	6
byte	0
pin	0
currState	TEST_LOGIC_RESET
nxtState	RUN_TEST_IDLE
firstState	SHIFT_IR
cmpRes	DONE
txBuffer	0x400694C8
txBuffer[0]	'3'
txBuffer[1]	'e'
txBuffer[2]	0
txBuffer[3]	0

Figure 10.14: tDataStructure after the command has been received.

After the TDI and TMS buffers were filled, they contain:

Watches 1

Name	Value
tdiBuffer	0x40068E90
tdiBuffer[0]	0x0
tdiBuffer[1]	0x7C
tdiBuffer[2]	0x0
tdiBuffer[3]	0x0

Figure 10.15: TDI buffer.

tmsBuffer	0x40068ED0
tmsBuffer[0]	0x30
tmsBuffer[1]	0x60
tmsBuffer[2]	0x0
tmsBuffer[3]	0x0

Figure 10.16: TMS buffer.

Remember that the data contained into the TDI buffer are reversed due to the LSB transmission direction. At least, this is what has been transmitted from the supervisor microcontroller.

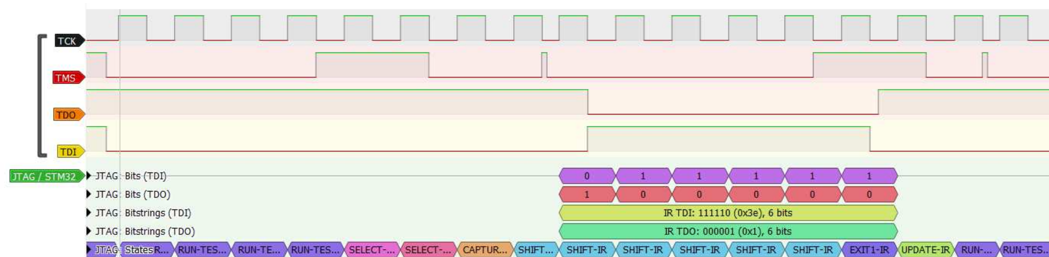

Figure 10.17: JTAG transmission.

Is possible to see the content of the instruction register shifted out, on TDO line, when the new instruction code is shifted in.

After some similar transmissions, if everything gone well, the DUT goes in test mode so the user can use the previously stored ATPG pattern to stimulate the DUT, reconfigure via JTAG the DUT again and send other patterns. The supervisor microcontroller will ensure the correctness of the output received from the DUT.

10.4 Improvements

This paragraph will introduce some improvement that may be considered.
Starting from the GUI discussed in chapter 9, it is just a sketch with bugs to fix and other features to add. For instance a panel can be added to the GUI that shows the state reached from the state machine when an instruction or data is transmitted.
Regarding the modules, the JTAG module could be improved by implementing the functions necessary to transmit the data or the instructions with the use of the GPIO. Although this solution is difficult to implement, it is the one that ensures the highest transmission frequency and the lowest memory demand. Having to respect the criteria of modularity and adaptability of the code, an idea could be the implementation of the transmission mode through GPIO as an alternative to the SPI, leaving to the user the choice on how to transmit the signals by means of a specific macro in the file *cnf.h*.
The ATPG module, on the other hand, can be further improved by implementing, as a parameter of the *send_pattern* command or as a stand-alone command, the possibility to choose where to move the ATPG patterns. Since the microcontroller used for the BurnIn+SLT platform does not have a flash memory, the received pattern cannot be saved inside it, so it can either be transmitted to the DUT or be transmitted to an external flash memory where it will be saved for future uses.

References

[22]Anand N, G. Joseph, S. S. Oommen and R. Dhanabal, "Design and implementation of a high speed Serial Peripheral Interface," 2014 International Conference on Advances in Electrical Engineering (ICAEE), Vellore, 2014, pp. 1-3, doi: 10.1109/ICAEE.2014.6838431.

[28]Barry, R. (2016). "Mastering the FreeRTOS™ Real Time Kernel."

[30]Bosch. (2013, 02 11). Application Note AN016. Tratto da https://www.bosch-semiconductors.com/media/ip_modules/pdf_2/gtm/gtm_ip_an016_mcs_spi_v03.pdf

[10]Brusca, M. (2014). "Tecniche di Burn-In nei System on Chip per applicazioni automotive." Palermo, Palermo, Sicilia.

[1]CERN. (11, 06). Introduction to Reliability Engineering. Retrieved from Indico Cern: https://indico.cern.ch/event/400079/contributions/956751/attachments/1184307/1716136/Introduction_to_Reliability_Engineering_-_CERN_06.11.pdf

[15]D. Appello and all, "Effective Screening of Automotive SoCs by Combining Burn-In and System Level Test"

[12]D. Calabrese, "System Level Test solutions for advanced automotive devices" (04/2019).

[2]H. Casier, P. Moens and K. Appeltans, "Technology considerations for automotive [automotive electronics]," Proceedings of the 30th European Solid-State Circuits Conference, Leuven, Belgium, 2004, pp. 37-41, doi: 10.1109/ESSCIR.2004.1356610.

[8]H. Yan, X. Feng, Y. Hu and X. Tang, "Research on Chip Test Method for Improving Test Quality," 2019 IEEE 2nd International Conference on Electronics and Communication Engineering (ICECE), Xi'an, China, 2019, pp. 226-229, doi: 10.1109/ICECE48499.2019.9058553.

[27]FreeRTOS. (n.d.). Tasks. Retrieved from FreeRTOS: https://www.freertos.org/RTOS-task-states.html

[11]JEDEC Solid State Technology Association. (2007). "Early Life Failure Rate Calculation Procedure for Semiconductor Components. In Early Life Failure Rate Calculation Procedure for Semiconductor Components."

[18]JTAG Connection Testing. (s.d.). Tratto da https://www.xjtag.com/about-jtag/jtag-connection-testing/

[23]Natarajan, B. (2018, 10 17). "What is SPI?" Tratto da Networking_Basic_LinuX: http://blmrgnn.blogspot.com/2018/10/spi-un1.html

[14] P. Bernardi, M. Restifo, M. S. Reorda, D. Appello, C. Bertani and D. Petrali, "Applicative System Level Test introduction to Increase Confidence on Screening Quality," 2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Novi Sad, Serbia, 2020, pp. 1-6, doi: 10.1109/DDECS50862.2020.9095569.

[3]Pinhong Chen and K. Keutzer, "Towards true crosstalk noise analysis," 1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051), San Jose, CA, USA, 1999, pp. 132-137, doi: 10.1109/ICCAD.1999.810637.

[13]S. Biswas and B. Cory, "An Industrial Study of System-Level Test," in IEEE Design & Test of Computers, vol. 29, no. 1, pp. 19-27, Feb. 2012, doi: 10.1109/MDT.2011.2178387.

[25]S. Tan and Tran Nguyen Bao Anh, "Real-time operating system (RTOS) for small (16-bit) microcontroller," 2009 IEEE 13th International Symposium on Consumer Electronics, Kyoto, 2009, pp. 1007-1011, doi: 10.1109/ISCE.2009.5156833.

[24]STMicroelectronics. (2015, September). ST. Tratto da ST: https://www.st.com/content/st_com/en/search.html#q=GTM-t=resources-page=1

[29]STMicroelectronics. (2016, 08). SPC5-STUDIO for 32-bit Power Architecture® MCU’s.

[7]STMicroelectronics. (s.d.). SPC58EC-DISP. Tratto da https://www.st.com/en/evaluation-tools/spc58ec-disp.html.

[6]STMicroelectronics. (s.d.). STM32F446RE. Tratto da ST: https://www.st.com/en/microcontrollers-microprocessors/stm32f446re.html

[26]Stoyanov, Y. (n.d.). RTOS Scheduling Algorithms. Retrieved from Open4Tech: https://open4tech.com/rtos-scheduling-algorithms/

[5]System-on-Chip Test Architectures. "Nanometer Design for Testability." (2008). In L.-T. Wang, C. E. Stroud, & N. A. Touba, System-on-Chip Test Architectures. Nanometer Design for Testability.

[21]T. Kirkland and M. R. Mercer, "Algorithms for automatic test-pattern generation," in IEEE Design & Test of Computers, vol. 5, no. 3, pp. 43-55, June 1988, doi: 10.1109/54.7962.

[9]"Testing of Embedded System." (s.d.). Tratto da NPTEL : https://nptel.ac.in/content/storage2/courses/108105057/Pdf/Lesson-39.pdf

[17]TI Semiconductor Group. (1997). IEEE std 1149.1 (jtag) testability primer. Tratto da https://www.ti.com/lit/an/ssya002c/ssya002c.pdf

[4]What is Crosstalk? (2018, 04 19). Tratto da Tech Web: https://techweb.rohm.com/knowledge/emc/s-emc/01-s-emc/6943

[19]Wikipedia. (2019, 11 03). Automatic test pattern generation. Tratto da Wikipedia: https://en.wikipedia.org/wiki/Automatic_test_pattern_generation

[16]Xun Jiang, Xiaoxin Cui and Dunshan Yu, "A JTAG-based configuration circuit applied in SerDes chip," 2011 9th IEEE International Conference on ASIC, Xiamen, 2011, pp. 707-710, doi: 10.1109/ASICON.2011.6157303.

[20] Y. Huang, W. Cheng, R. Guo, T. Tai, F. Kuo and Y. Chen, "Scan Chain Diagnosis by Adaptive Signal Profiling with Manufacturing ATPG Patterns," 2009 Asian Test Symposium, Taichung, 2009, pp. 35-40, doi: 10.1109/ATS.2009.36.