UNIVERSITÀ
DEGLI STUDI
DI PALERMO

NDS
NETWORKING AND DISTRIBUTED SYSTEMS
RESEARCH GROUP

Implementation of cost effective solutions for testing in automotive environment

Tesi di Laurea Magistrale in Ingegneria Informatica

Emanuele Di Miceli

Relatore: Prof. Daniele Peri
Correlatore: Ing. Francesco Camarda STMicroelectronics.

UNIVERSITÀ DEGLI STUDI DI PALERMO

DIPARTIMENTO DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Implementation of cost effective solutions for testing in automotive environment

TESI DI LAUREA DI
Emanuele Di Miceli

RELATORE
Prof. Daniele Peri

CONTRORELATORE
Ch.mo Prof. Marco La Cascia

CORRELATORE
Ing. Francesco Camarda
STMicroelectronics

ANNO ACCADEMICO 2019 – 2020

UNIVERSITÀ DEGLI STUDI DI PALERMO
FACOLTÀ DI INGEGNERIA

LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Implementation of cost effective solutions for testing in automotive environment

Tesi di Laurea di
Emanuele Di Miceli

Relatore:
Prof. Daniele Peri

Controrelatore:
Ch.mo Prof. Marco La Cascia

Correlatore:
Ing. Francesco Camarda
STMicroelectronics

Abstract

The present work is the synthesis of the internship period carried out at STMicroelectronics (Palermo) within APG Automotive Digital Solution Burn-In group.
In the last decade the electronics has progressively increased its presence and importance in the automotive business. The average car embeds electronics for: safety, Advanced Driver-Assistance Systems(ADAS), power management, train control system, infotainment and so on. Safety and security are two fundamental prerequisites, so the electronics must comply with high standard; this means automotive grade should ideally be free of defects. In this context the role of testing becomes more and more important.
In order to prevent defective products from entering the market, these devices unlike consumer electronics, must be tested during the various stages of production. For instance, a malfunction in a vehicle's air-bag activation system may have serious consequences for the passenger health. In this thesis the versatility of the automotive microcontrollers (*SPC58 family* manufactured by STMicroelectronics) is explored for the implementation of cost effective solutions for the testing of automotive digital devices.

Contents

Introduction

The automotive industry involves the design, development and manufacturing of vehicles. An electronics device is certified automotive grade only if it satisfies certain quality manufactory standards. Some of these are IATF 16949, AEC-Q100 and AEC-Q200. They guarantee the reliability of the automotive devices even under extreme environmental conditions. Obviously, since all these requirements have to be added to the production chain, the cost of an automotive device is on average 30% higher than the consumer equivalent. For example, an automotive microcontroller like the SPC58EC80E5 [1] is sold for about 20$, while the consumer device STM32F446RE [21] is priced around 5$.
Before being placed on the market, all automotive devices must be tested in several steps: from the production of silicon wafers up to the last functional tests like Burn-In. Usually the testing is managed by expensive devices (based on LINUX, for instance) which are connected to multiple Device Under Test (DUT).
In this thesis the traditional testing equipment is replaced with an automotive microcontroller.
An MCU - unlike a traditional CPU - include several peripherals within a single chip, indeed these devices are designed to interact (control) with the surrounding environment. Microcontrollers are exploited in areas like: IoT, robotics, control systems, automotive and so on.
The low cost, versatility motivate the choice to propose testing solutions based on microcontrollers. From this perspective, an MCU becomes the supervisor and it's able to manage the testing of other integrated circuits. In particular, this thesis refers to a previous work [24], in order to integrate it with additional features.
The first chapters introduce the reader to the topics discussed in the thesis. Chapter one provides a general overview of microcontrollers and their context of use, referring to the automotive environment: ABS, ASR, AIRBAG, ECU. This chapter highlights how these devices are different from a traditional CPU, both in terms of the number of peripherals directly integrated into a single chip, and in the significant difference of the computational resources. This last consideration

underlines how profoundly different are the software design techniques; the proposed solutions are consequently aimed to a careful and scrupulous use of MCU resources.
The third chapter introduces some of the testing methodologies included in the state of the art; techniques like Burn-In, ATPG, and JTAG will be analyzed. The thesis refers to the previous, proposing solutions that can be managed directly by a microcontroller. Eventually the next chapter concludes the first part, it provides an overview of the hardware and software tools used. The entire source code is written in ANSI C language, within the SPC5-Studio development environment. This last is not a conventional tool: it includes APIs and other essential features for writing applications that can be run by the *SPC5 family* microcontrollers (produced by STMicroelectronics). In addition the logic analyzer, a device that allows to capture digital waveforms, is used together with the analysis software PulseView.
Once all the basic notions, necessary to understand the thesis work have been introduced, chapter four deals with the description of the implementations. It is divided into two parts: the first one discusses a mode of vectorial communication, based on the parallel control of a General Purpose Input/Output (GPIO) port. In the second part instead proposes the integration of a Pseudorandom Number Generator (PRNG) algorithm developed at the Politecnico di Torino, it is exploited to generate stimulation patterns for testing. The chapter describes two operations: sending patterns and checking the test result.
In conclusion the first of the last chapters introduces the Generic Timer Module (GTM): a peripheral designed by BOSCH (as an Intellectual Property) and integrated in some automotive microcontrollers. The peripheral is used to control the vehicle's mechanical systems, a practical example is the engine carburation management. The chapter offers a basic device overview, useful to understand the application described in the following. The purpose of the application is to provide an additional communication interface to the microcontroller, similar to a Serial Peripheral Interface (SPI) based on the GTM.
Finally in the conclusions the proposed solutions are discussed, analyzing their strengths and weaknesses.

Chapter 1

MicroController Unit (MCU)

This chapter is an overview of microcontrollers in order to introduce these devices and the applications in which they are used. It's important to underline that microcontrollers are different from a traditional CPU and therefore the approach to software design executed by these devices is totally different. CPUs and MCUs are not alternative devices, but are used in different contexts in which one could not replace the other. The chapter starts with a general description of the MCUs and the automotive environment: hardware features and applications in which is possible to find them. After this brief presentation will be deepened all the topics that later will be the subject of this thesis work.

1.1 MicroControllers Overview

An MCU [4] is a single device containing in addition to a traditional CPU a number of peripherals, I/O devices, memories and analog components. The systems where is possible to find a microcontroller are among the most varied: domotics, industrial machines, automotive, control systems, robotics, medical devices, IoT, and so on. One of the major difference between a general purpose CPU and an MCU is the number of peripherals integrated in it, this is why a microcontroller is designed to interact with the physical world surrounding it, in which it is embedded. For example an MCU inside a washing machine may need to communicate with the device that regulates the flow of water. A general purpose CPU is definitely a more powerful device in terms of computational resources, however in the context of the use of microcontroller other features are required. Often these applications require high performance in terms of energy

saving, with consumption also of the order of nanowatts. This can be achieved by setting the devices at low working frequencies (order of kHz as well), but still sufficient for the purpose of the application. In addition, in many real time systems a strict respect of time constraints is mandatory, a common CPU also with much more computing power is not designed to ensure this kind of constraints. The Table 1.1 shows some of the main differences between CPUs and MCUs.

MCUs	**CPUs**
Special purposes applications like Embedded System: automotive, domotics, IoT, robotics, medical devices.	General purpose applications like PC, laptop, smartphones.
Real Time Operating System (RTOS), *bare-metal*, FORTH interpreter.	General purpose OS like Windows, Linux, macOS.
Low clock speed kHz order as well.	High clock speed GHz order.
Low computational capacity.	High computational capacity.
Low cost system.	High cost system.
Low power consumption.	High power consumption.
4 - 32 bit architectures.	32 - 64 bit architectures.
A lot of peripherals and memories embedded into a single chip.	Only CPU. RAM, ROM and other peripherals are connected externally.

Table 1.1: Some differences between MCUs and CPUs.

A microcontroller is usually a cheap device, the cost for each unit may also not exceed 1$ as well. The low cost and the versatility of use in multiple contexts have allowed a great success on the market, in 2002 about the half of the global market of CPUs sold were MCUs [22]. As stated above, microcontrollers interact with the surrounding physical environment, so they must be able to respond to the stimuli provided by it. The concept of interrupt is critical, this latter are raised to report that a certain event has occurred, for example a new data available from a sensor. The interrupts are important in a context of energy saving because the MCU can remain inactive, thus save energy until an interrupt is raised. Typically for an application a several number of MCUs are needed, each of which deals with a part of the physical environment. An application is basically a black box that receives external stimuli via sensors, while performs actions through actuators.

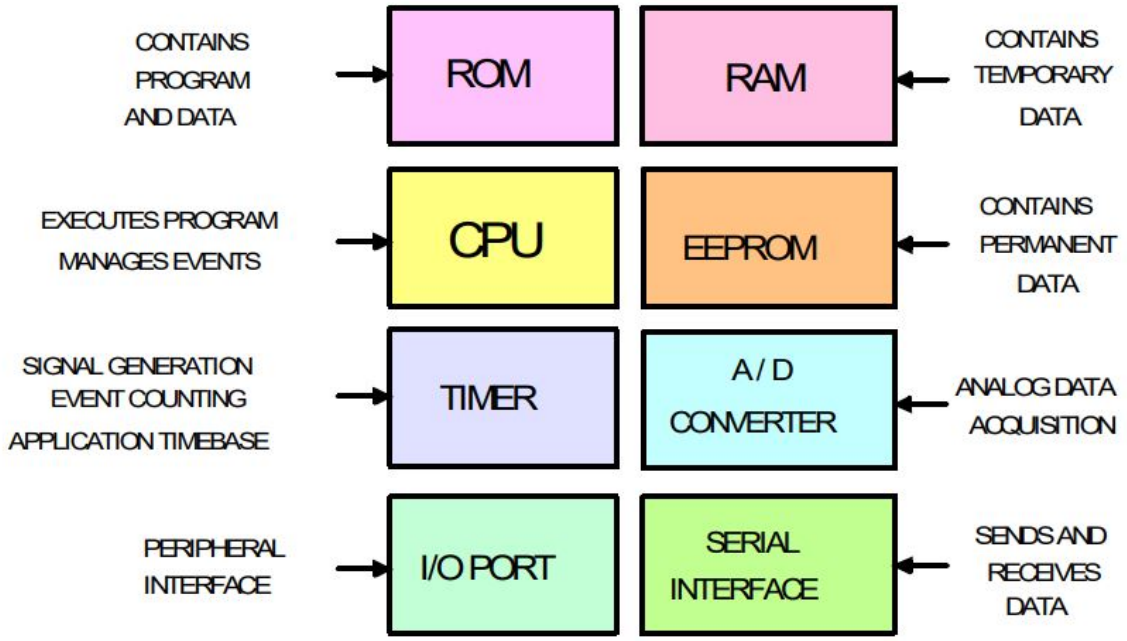

Figure 1.1: Inside an MCU. Image from [4]

As a part of an applications, the MCUs are then included within this chain, in fact they are connected to sensors and actuators, the first to process external inputs, and the second to perform actions on the environment. In order to do this, an MCU is rich of I/O peripherals like: SPI, I2C, GPIO, Ethernet, USB, and so on. A versatile peripheral is surely the GPIO, each pin is configurable by software as input, output or alternate function. The MCUs usually included ADCs, which are devices able to convert an analog signal to a digital, this is necessary because the environment is intrinsically analog but it is processed by the microcontrollers as digital. Some MCUs integrate components to make special purpose calculations, for instance a DSP processor for audio signal and image processing, speech recognition and so on. Not least the timers can measure events or be used for generating PWM signal for the control of systems such as engines.

In summary as shown in the Figure 1.1 an MCU usually includes :

- CPU from 4 bit to 32 bit processors;
- RAM memories;
- Clock generators (quartz oscillator, RC circuit);
- Non-volatile memories: Flash, EEPROM, ROM, EPROM;
- PWM generators and timers;
- Several communication interfaces: UART, I2C, Ethernet, USB, SPI;
- ADC and DAC converters;
- JTAG for debugging.

For the software development on a microcontroller, it is typically used an environment (executed on a development machine) capable of compiling the object code for the target MCU (cross-compilation). Often this environments provides APIs called HAL, these software libraries allow the user to configure and use the device and its peripherals in different ways. An example of a development environment is SPC5-Studio, reference software for this thesis work described in the third chapter. Nevertheless is also possible for the MCU to run language interpreters based for instance on Python or FORTH. The choice always is closely related to the hardware resources of the device, indeed it may have no problems running the simple FORTH interpreter, while more complex Python-based environments may not be supported.
On some microcontrollers that have a sufficient computing capacity can be executed particular OS called Real Time Operating System (RTOS). In short they are simple systems capable of ensuring strict operational constraints. Eventually when an MCU does not run any operating system the development is defined as *bare-metal*.

1.1.1 Automotive Microcontrollers

In the automotive industry, integrated circuits such as microcontrollers are applied for vehicle management. The MCUs can be found in different contexts: from the control of the carburetion up to the management of the entertainment; more subsystems can communicate and cooperate with each other (Figure 1.2).
Often inside a vehicle some MCUs are replicated, usually there is a secondary microcontroller that verifies the operation of the primary one, so as to always ensure the critical functionality. Moreover, certain multi-core devices have a shadow core: it replicates the operations of the main core, for example with a delay clock cycle. Then, the verification of the correctness of the performed operations is always possible. Among all the devices integrated in a vehicle the fundamental one is definitely the ECU. Such system works in real time, continuously monitoring the sensors connected to it and responding to stimuli through its actuators, controlling: cooling system, emission control, fuel injection, diagnostics, ignition system, and so on. A correct setting of the ECUs ensuring maximum performance in terms of safety and consumption. Another use case of automotive MCUs is certainly the handling of safety systems, among the most popular: ABS, ASR, BAS. A practical example is the activation of the ABS system: the sliding of the tires on the slippery asphalt can be detected by a microcontroller through a sensor, and consequently the ABS system is activated by its actuators.

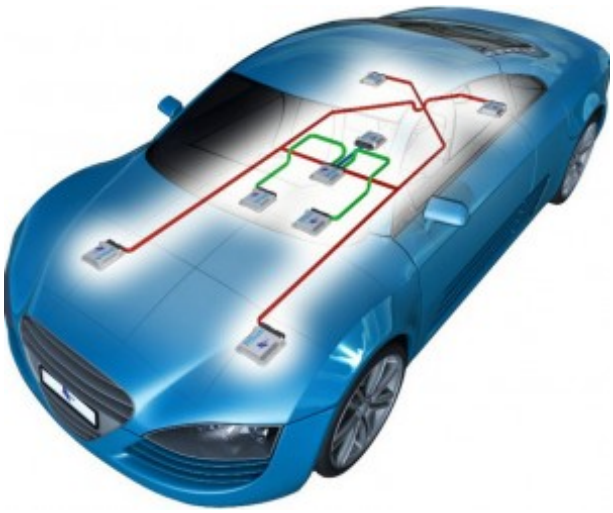

Figure 1.2: A series of microcontrollers inside a car. Image from [11]

With the spread of IoT and the growing development of autonomous or semi-autonomous driving even more frequent use of MCUs is needed. Moreover these ICs manages passenger comfort, air conditioning, navigation systems, infotainment and not least entertainment.
It's important to note that more a vehicle becomes integrated and connected, more it is an attack surface for hackers, hence is crucial to design reliable and robust systems also from this point of view. The testing of these devices is crucial, they are designed to have the minimum number of failures. Later in this thesis this topic will be widely discussed.

1.2 Serial Peripheral Interface (SPI)

The SPI [31] is a serial and synchronous bus interface used in the communication of integrated circuits such as memories, MCUs and other digital devices. The transmission is controlled by a master device that emits the clock signal, one or more slave devices are connected to the master in different configurations. The bus is composed of four signals

- Master Input Slave Output (MISO);
- Master Output Slave Input (MOSI);
- Serial Clock (SCK);
- Slave Select (SS).

As previously mentioned, the communication is serial therefore the bits are sent one at a time, and it is also synchronous because among the signals that make up the bus there is a clock signal (emitted by the master) that scans the transmission. The communication can take place simultaneously in both sending and receiving, then the bus is defined as full duplex. Both master and slave devices are equipped with a shift register typically of eight or sixteen bits. At each clock pulse the devices that communicate emit a bit from their shift register and replace it with a bit emitted by the others. Specifically as shown in the example in Figure 1.3, a bit emitted by the master in the MOSI line will replace the corresponding bit in the shift register of the slave device, and at the same time the bit emitted by the slave in the MISO line will replace the corresponding bit in the shift register of the master device.

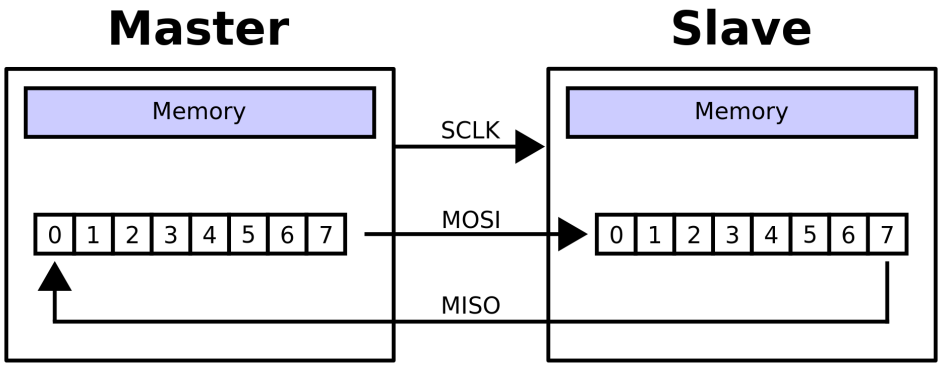

Figure 1.3: SPI transmission schema. Image from [38]

The two parameters CHPOL and CHPHA determine the polarity and the phase of the clock respectively. The CHPOL value defines the logical level of the clock line when the transmission is not active, while the CHPHA parameter specifies the sampling edge of the data line by the receiver: rising or falling edge of the clock signal. With two parameters four different configurations are available, but two of which are equivalent (CHPOL = 1 CHPHA = 1 and CHPOL = 0 CHPHA = 0).

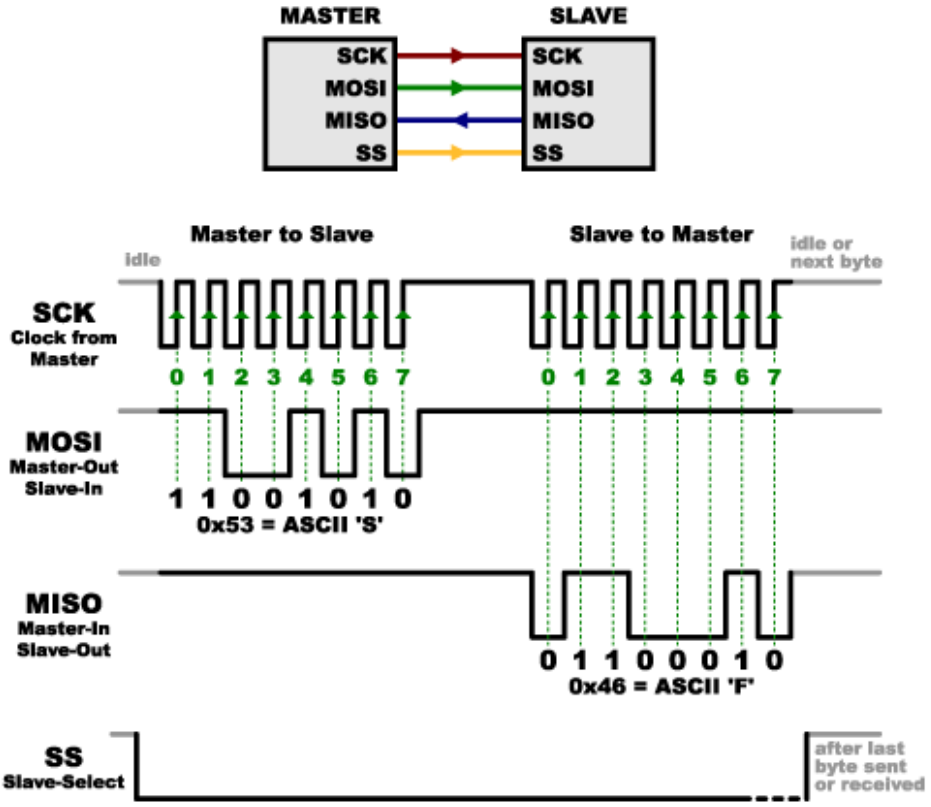

Figure 1.4: Example of an SPI transmission. Image from [15]

The Figure 1.4 shows a simple transmission between a master and a slave device, the data exchanged are ASCII characters. Note that the SS line enables the device with which to communicate, in some contexts can also be omitted. It's important to note that the SPI standard does not define a limit transmission rate, however this dependent on the communicating devices.

1.3 General Purpose Input/Output (GPIO)

A GPIO is a communication interface integrated in microcontrollers and others integrated circuits to read or write digital signals (voltage values encode the logical value 0 or 1). Usually the GPIOs are used to communicate with external devices such as sensors, memories and so on. This interface can be set to input, output and even alternative functions (provided by internal peripherals like SPI) by writing appropriately the configuration registers. The settings typically include also the electrical characteristics of the peripherals such as pull-up or pull-down configurations. Often different devices operate at several voltage levels (typical values are 2V, 3.3V, 5V) and therefore it may be necessary to connect the GPIOs using appropriate level shifters to ensure compatibility. Generally the GPIOs are grouped in ports and configurations are available for a set of pins belonging to the same port. Finally this peripheral can raise interrupts or used in Direct Memory Access (DMA) mode in order to minimize the overhead of the CPU.

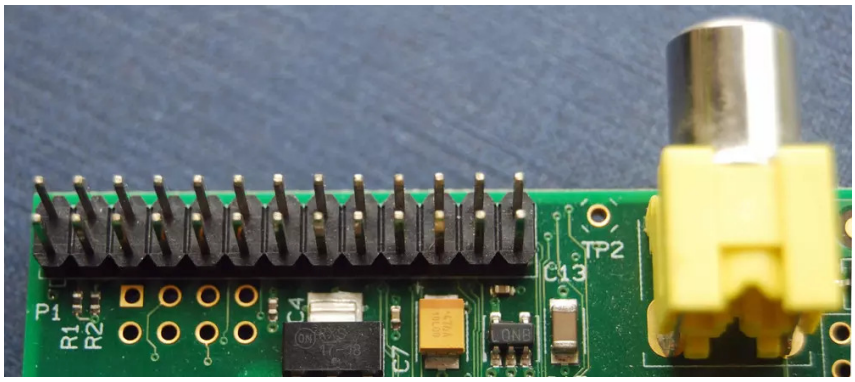

Figure 1.5: An example of a GPIO port.

1.4 Direct Memory Access (DMA)

The I/O management is a critical feature, an MCU interacts with the external environment through the peripherals in different ways. Clearly the instructions executed to manage the I/O consume processor time, hence there are several techniques to efficiently and independently perform this task. The following are three common ways in which an MCU can handle read and write from peripherals:

- Busy Wait;
- Interrupts;
- DMA.

The first technique involves a loop in which the CPU remains until the data from a peripheral is available. Basically in this case the CPU remains locked and cannot executes other instructions, so the waste of clock cycles is maximum. The second technique uses the interrupts, the CPU executes other instructions, when the peripheral is ready it raises an interrupt and consequently the CPU stops executing the code it was executing to jump to an interrupt routine that handles the I/O operation. Even if the microcontroller can execute other instructions while waiting for data from the peripheral, when interrupts become too frequent the resulting overhead could degrade the performance of the application.
DMA [3] is an integrated device used to perform CPU-independent memory access operations, this frees the CPU because the DMA controller is able to control the data bus independently. Just an operation has to be executed by the CPU, the setting of the DMA registers to initialize the I/O task. When the data transfer is completed the DMA controller raises an interrupt to report the event.

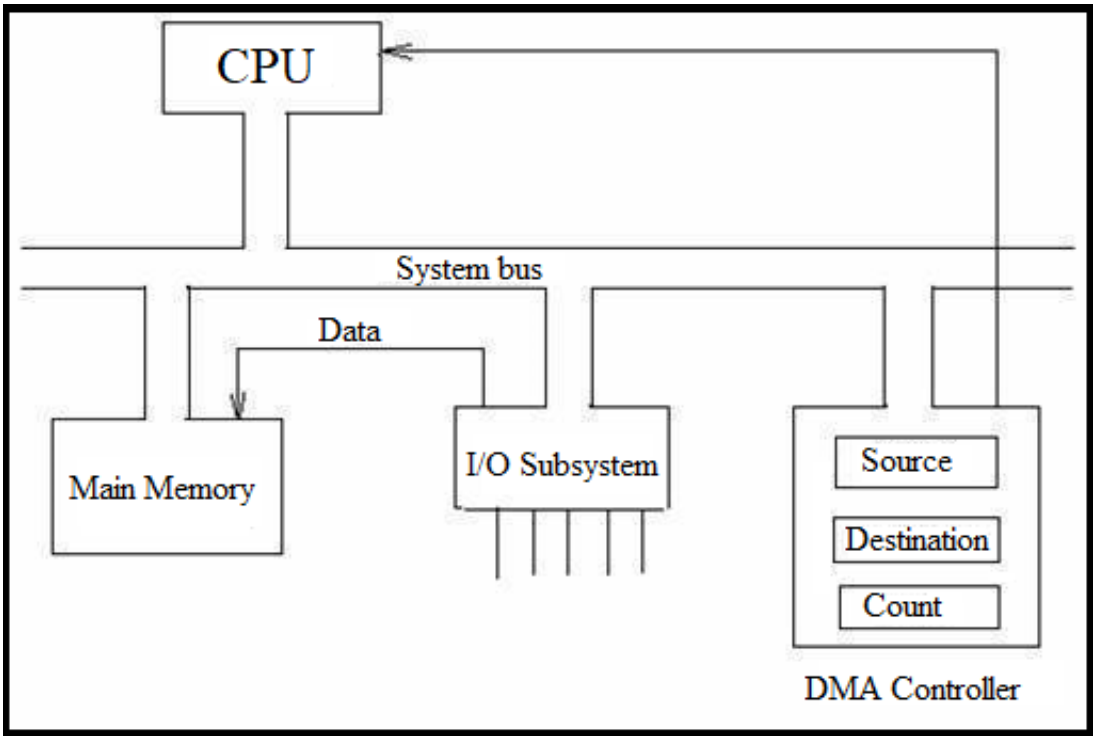

Figure 1.6: A schema of a DMA system configuration. Image from [35]

The DMA system (Figure 1.6) is configured via software by writing configuration registers that usually are four:

1. Inside the first register (*Source*) is written the address of the device from where to read or write the data;
2. The second register (*Destination*) contains an identifier for the I/O peripheral to use;
3. The third register (*Count*) stores the amount of data (in bytes or words) to be transferred;
4. The fourth register reports the completion of the I/O operation.

It's important to note that even if the DMA relieves the CPU from the I/O management the data bus is still occupied, then the CPU may not be able to use the bus until the operation is completed. Some transfer modes are the following [8]:

- Burst Transfer: the data bus is busy by the DMA controller until the data transfer is completed;
- Cycle Stealing: the transfer is performed only when the peripheral is ready, therefore the bus occupation by the DMA controller is not continuous, but can be interrupted if the peripheral does not have new data;
- Transparent/Hidden: in this case the controller can just use the bus if the CPU is not occupying it.

1.5 Real Time Operating System (RTOS)

An Operating System (OS) provides the basic functionalities to handle the hardware and software resources of a computer system. Among the essential components of an OS for example there are the Scheduler, which selects the running processes, or the File System which manages the stored files. However a General Purpose OS like Linux or Microsoft Windows is not always suitable for all contexts of use, usually when dealing with MCUs there is a need to have a class of systems able to run in poor computational resources environments, and capable of ensuring strict time constraints. A conventional OS can crash at any time, this may not be tolerated in particular applications. As an example, a crash in the flight control system of an aircraft can lead to catastrophic consequences.
An RTOS system is essentially a scheduler able to guarantee a deterministic execution time for the instructions executed by a microcontroller. This type of systems does not include all the components of a general purpose OS, this is why the devices on which they are executed don't have the necessary computational resources. Moreover many of these components are useless in these contexts, the Table 1.2 summarizes the main differences between them.
FreeRTOS [36] is a Real Time Operating System (RTOS) available under MIT license and free to download from the developer's website. The system provides methods for creating threads (also called tasks), mutex and semaphores for their communication. In addition it assigns a running priority to each of the threads, this is the mechanism that allows the scheduler following

a round-robin order to ensure time constraints. As mentioned above, to assure low memory occupancy and running speed, the system does not integrate some of the advanced features present in a normal OS, furthermore the *tickless kernel mode* can be activated for low-power applications. Some add-ons are available, for example the FreeRTOS command line interface. Further details are available in the reference manual. [33]

Real Time Operating System (RTOS)	**General Purpose Operating System (OS)**
Deterministic time of execution.	Random time of execution.
Low amount of computational resources needed.	High amount of computational resources needed.
Simple user interface like CLIs.	Complex user interface like GUIs.
Connectivity is provided as add-ons.	Full connectivity support.
Special purporse applications like robotics, automotive, flight control and so on.	General Purpose applications: laptop, smartphones, PC.
Static memory allocation.	Dynamic memory allocation.
No Virtual Memory.	Virtual Memory.

Table 1.2: Some differences between RTOS and General Purpose OS.

Chapter 2

Automotive Testing: State of the Art

The design, development and sale of motor vehicles are part of the automotive industry. In this context electronic systems [10] handle vital functions for a vehicle: check the braking system, air-bag activation, assisted braking and so on. The testing of these systems is a crucial objective to prevent defective products from entering the market. The cost for testing a device is considerable both in terms of time and money, therefore it's important to develop efficient and effective testing approaches. Today the industry is oriented on the concept of Design for Testability (DFT): the ICs are designed to integrate already the circuitry necessary for testing. The complexity of the devices and an increasing demand for lots from the market, make obsolete and inefficient the manual techniques that were used in the past. The following will describe some of the most known testing methodologies in the automotive field. The first part of the chapter introduces DFT, while the second describes the JTAG standard, which basically provides an universal access door to the device. Finally testing methodologies such as ATPG, Burn-In and System Level Test (SLT) are presented. From now on the tested device will be referred to as Device Under Test (DUT).

2.1 Design for Testability

In VLSI systems the number of transistors and then logical ports that are integrated within an Integrated circuit (IC) has grown enormously. The performance of these systems continue to improve, but on the other hand the complexity has become such that manual tests are inefficient. Design for Testability [39] is a set of techniques and guidelines useful to design devices in order to make testing operations more efficient and less expensive. These techniques can be divided into two categories:

- Ad hoc: metologies developed for a particular device and not always applicable in a generic way;
- Structured: a set of general strategies applicable on devices that are designed to implement them. Some of the most known are: JTAG, ATPG, BIST, Burn-In.

Testing a device means verifying that it does not present any defects that could lead to malfunctions during its use. A fault model is able to provide a general metric for the error evaluation, clearly these models do not include all possible defects. Nevertheless the alternative is to consider a DUT from an electrical point of view: a set of resistors, capacitors, inductors and so on [42]. Often this mode is impractical because already for a simple AND port the amount of defects that is possible to model grows significantly.
The STUCK-AT model is one of the most known, it is based on the hypothesis that in the case of fault the input or the output of a logic port is stuck at a fixed high or low logical level. Other examples are the following:

- Cellular Fault Model;
- Hard Bridging Faults;
- Transistor Faults;
- Parametric Faults.

It's important to underline that a model is just an abstract representation of the faults, for instance in the case of the STUCK-AT the cause is not distinguished, the logical level of a line could be blocked due to a short-circuit to mass or a voltage reference.

The quality of a model is measured according to the amount of fault that it is able to detect: the so-called fault coverage. However even a fault coverage of 100% does not guarantee the absence of defects, indeed the fault coverage is referred to the model which is always an imperfect abstraction [26].

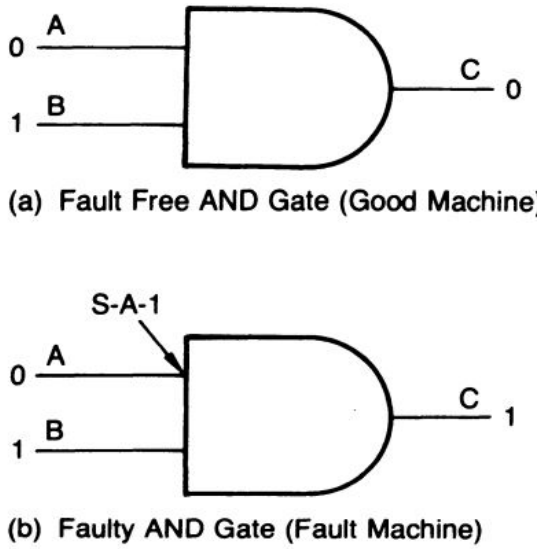

Figure 2.1: STUCK-AT model example. Image from [39]

The Figure 2.1 shows a simple example of the previous, the gate 2.1(a) is fault free while the gate 2.1(b) is faulty. A possible test could be the stimulation with the input pattern ($A = 0, B = 1$). As it's clear the input A in 2.1(b) is "STUCK-AT-1", then the port is defective since with this input pattern the output C should have the logical value 0 and not 1.
Generally a test model works if it's possible to distinguish whether a machine is functioning or not (Figure 2.2).

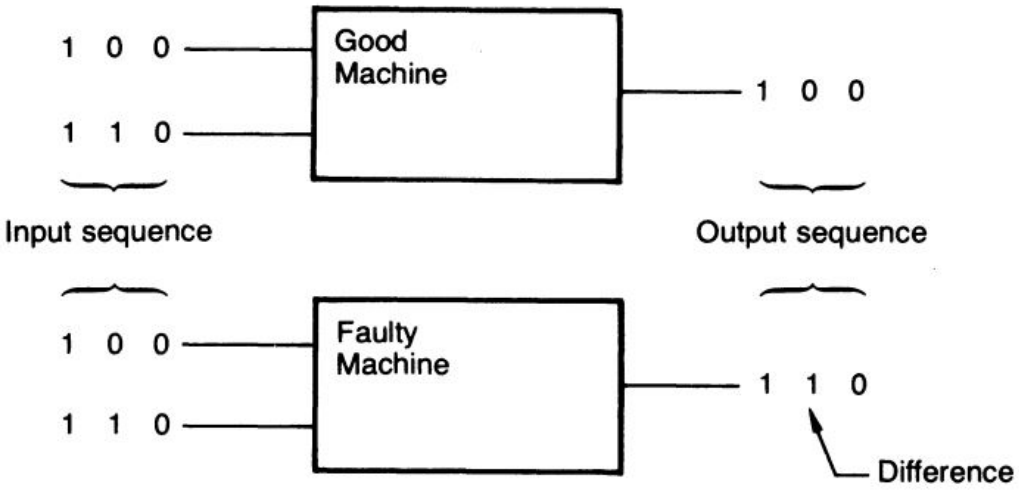

Figure 2.2: Faulty and Good machine. Image from [39]

Controllability and *Observability* are the principles on which the *Design for Testability* is based. The example in Figure 2.1 is also useful to clarify these two concepts. To check if the input *A* is "STUCK-AT-1", the first of the two properties is required. Indeed the inputs *A* and *B* are controlled to perform the simple test described above. Also the second property is necessary because the output *C* must be "observable" to verify the test result. These two features are not trivial, in fact in a complex system such as a sequential network may not be fully guaranteed.

2.2 Joint Test Action Group (JTAG)

JTAG [14] is an industrial standard (IEEE 1149.1) developed by the *Join Test Group*, a consortium of integrated circuit manufacturers. Devices implementing the standard are equipped with additional circuitry that provides testing capabilities. Via JTAG a device is accessed in *test mode*, then a DUT can be configured in all its parts, also some additional features (hidden in *user mode*) are available. For example the device's RAM and flash memories can be written or read even while it is executing instructions, as well as all configuration registers. The standard provides several signals, some are mandatory while others depending on the particular device and the manufacturer. These last along with the precise sending sequence, are usually confidential to avoid that a normal user has universal access to a device. The mandatory lines are the following:

1. Test Clock (TCK);
2. Test Mode Select (TMS);
3. Test Data In (TDI);
4. Test Data Out (TDO).

Before analysing each of the above, it's necessary to introduce the JTAG Test Access Port (TAP). The previous (Figure 2.3) is a state machine whose purpose is to manage the device in *test mode*, it consists of a series of registers and multiplexers.

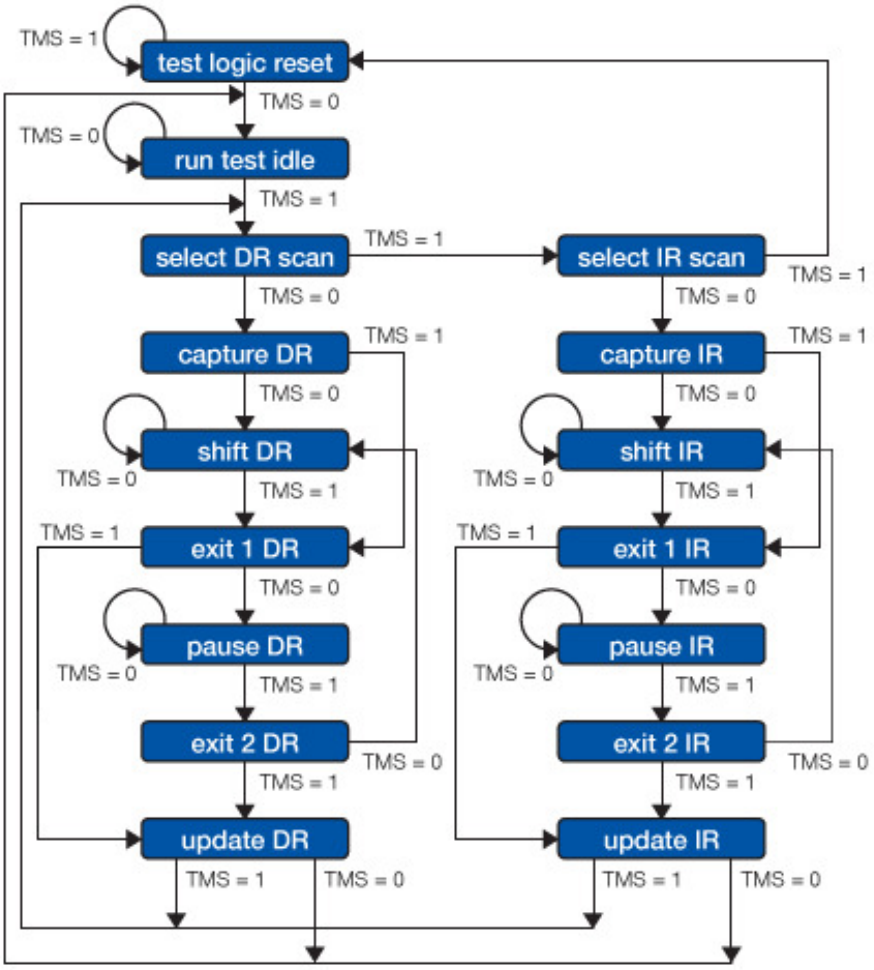

Figure 2.3: JTAG TAP. Image from [32]

As shown in the Figure 2.3 the TAP is composed of sixteen different states, to move inside the state machine is used the TMS signal, sampled on the rising edge of the TCK signal. Some states are symmetrical and refer to different shift Registers: Instruction or Data Registers. These will be analysed later in the course of the paragraph.

- Test-Logic-Reset: in this state the device executes normal instructions, all test modes are disabled. In addition there is an optional fifth Test Reset (TRST) line that allows to reach this state by keeping the TMS signal high for 5 cycles of the TCK signal [41];
- Run-Test/Idle: the machine performs the selected testing operations;

- Select-DR-Scan , Select-IR-Scan: the initial state for accessing Data or Instruction registers;
- Capture-DR , Capture-IR: the first state allows the selected data register to shift data (input or output), the operations are performed on the rising edge of the TCK signal. The second state is used to load parallel inputs into the instruction register;
- Shift-DR , Shift-IR: in both states the same operation is performed on the selected data register or on the instruction register: a bit coming from the TDI signal is shifted within the register at each rising edge of the TCK signal, and consequently one is emitted on the TDO line in correspondence to the falling edge of the same signal;
- Exit1-DR , Exit1-IR: these states end the shifting operation for the respective registers, the next reachable states are PAUSE or UPDATE;
- Pause-DR , Pause-IR: both used to pause shifting operations in their registers;
- Exit2-DR , Exit2-IR: they end shifting and update the corresponding registers;
- Update-DR , Update-IR: the data contained in the selected data register or the instruction register are loaded into a latched parallel output.

The instruction register contains the type of test to be performed or the address of the data register to be accessed, the standard provides the following 3 mandatory instructions, however some devices may include other:

- BYPASS: the TDI and TDO lines are connected through the BYPASS register (an one-bit register), the instruction is useful to control several DUTs simultaneously;
- EXTEST: the effect of this instruction is to connect the Boundary Scan Register (BSR) to the TDI and TDO signals, so is possible to read or set the state of the device pins;
- SAMPLE/PRELOAD: like the previous also this instruction allows to connect the TDI and TDO signals to the BSR register. Nevertheless in this case the instruction takes a « snapshot » of the device during normal operation. It's also possible to preload test data within the BSR register before executing the EXTEST instruction.

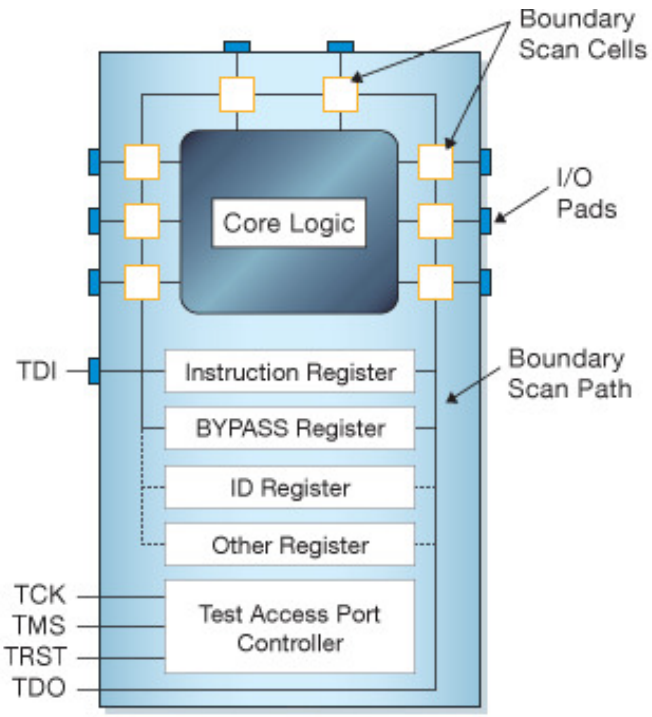

Figure 2.4: Schematic Diagram of a JTAG enabled device. Image from [32]

IDCODE is another common instruction, the latter is useful to get feedback from a device, the TDI and TDO signals are connected to the IDCODES data register (if any) in order to read the device ID. The standard requires that the TAP necessarily includes two data Register: BSR and BYPASS, however as in the previous case, one device may include others.

- Boundary Scan Register (BSR): as shown in the Figure 2.4 this register consists of Boundary Scan Cells and it is connected to the device I/O pins to test their proper functionality;
- BYPASS: this one-bit register combines the TDO and TDI lines to easily manage multiple connected devices;

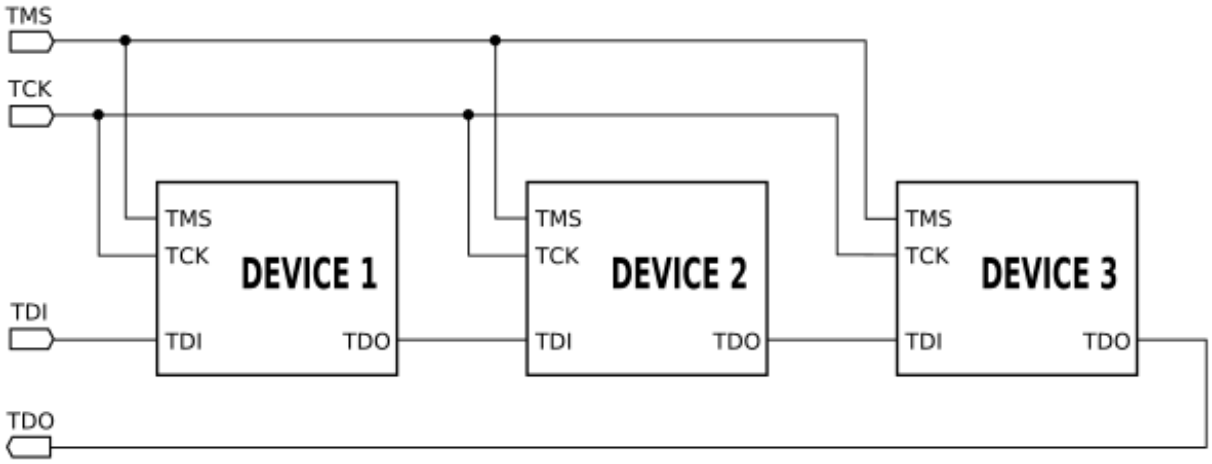

Figure 2.5: Daisy Chain connection. Image from [37]

Finally connecting multiple DUTs in cascade mode (*daisy chain* connection) it's possible to drive the TAPs of heterogeneous devices with the same set of signals. In this way as the Figure 2.5 shows, more devices can be tested at the same time. This feature becomes useful as only one supervisor can manage more DUTs.

2.3 Automatic Test Pattern Generation (ATPG)

ATPG is a testing strategy whose purpose is to stimulate a DUT with particular bit patterns which model possible defects of a device. By analyzing the DUT's response to stimulation, any defects, together with various information on them can be identified.
A DUT is configured to run ATPG test through JTAG, it's important to note that JTAG can be considered as an universal access port to the DUT, while ATPG instead is a testing mode. Usually an ATPG transmission includes at least four signals:

1. Scan In (SI): to send stimulation bit patterns to a DUT;
2. Scan Out (SO): line encoding device response to stimulation;
3. Scan Enable (SE): enable testing;
4. Serial Clock (SCK): provides the clock signal needed to scan operations.

During this test a device ends to be an IC and simply becomes a sequential network to be stimulated by ATPG pattern and from which to observe the response.
As shown in the Figure 2.6, inside a DUT a sequence of so-called Scan Flip-Flop (SFF) makes up a scan-chain. These flip-flops are connected to the combinational part of the IC composing the sequential network that represents a device in ATPG mode. A scan-chain behaves like a shift-register, within the latter the bits of an ATPG pattern are shifted. Moreover different scan-chains test multiple areas of a DUT.

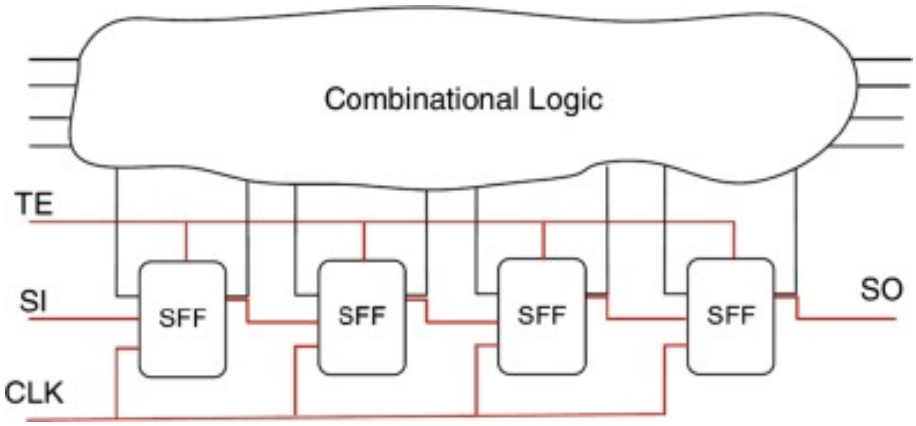

Figure 2.6: A scan-chain example. Image from [6]

The method consists of two distinct phases:

1. *Shifting phase*: at each pulse of the SCK signal the SFF flip-flops are filled via SI line with the bits of an ATPG pattern, since each flip-flop is connected to the combinational network, the latter will evolve on the basis of the data stored in the flip-flops;

2. *Shifting + capture phase*: in this phase the evolution of the sequential network (SFF plus combinational network) is stored in the flip-flops and then shifted out through the scan-out line. Since the device's response is shifted out on the scan-out line, it's possible to analyze it in order to verify the presence of faults in the device.

As briefly mentioned above, an ATPG pattern is a sequence of bits designed to make visible any defects of a DUT after its stimulation. These patterns are usually generated by appropriate software integrating algorithms that exploit mathematical models often based on heuristic techniques: D, PODEM, FAN are among the most known algorithms [23].
Another possibility to generate ATPG patterns is based on pseudorandom numbers, using a Pseudorandom Number Generator (PRNG) algorithm to build seemingly random but repeatable (knowing the initial seed) bitstream. An example of this latter technique will be widely discussed in this thesis.

2.4 Burn-In

Burn-In [26] is a testing methodology whose principle is to make evident extrinsic defects of a device. This kind of faults are not caused by a bad design, but rather due to defects during the production phase.
The test is based on the *Bathtub curve* (Figure 2.7), on the x-axis is depicted the lifetime of a device, while on the y-axis is shown the failure rate (which can be measured according to different criteria) [12]. Inside the lifecycle of a device are distinguished three phases corresponding to the areas of the Bathtub curve:

1. Infant mortality;

2. Constant failure;

3. Wear out.

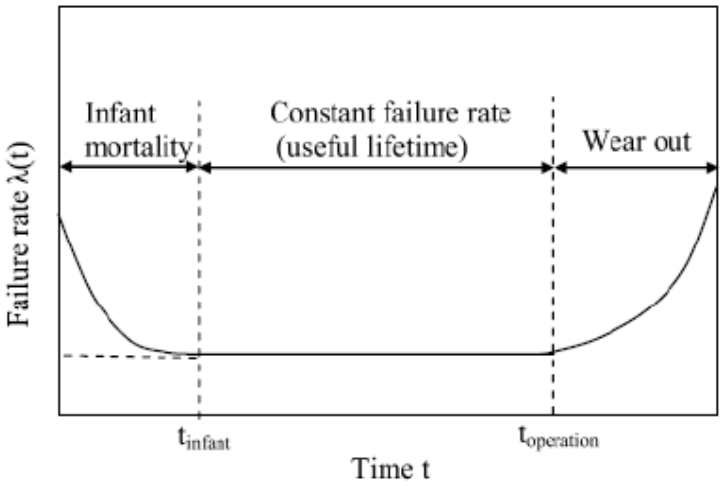

Figure 2.7: The Bathtub curve. Image from [40]

How the Figure 2.7 shows, the shape of the curve (the failure test rate) decreases at the beginning (*Infant mortality*) and grows at the end of the life cycle (*Wear out*), while it remains approximately constant in the middle (*Useful lifetime*). It's intuitive to motivate the previous behavior: if a device is defective then the defects will lead to a failure already during the first period of use; in the central part of the curve (*Constant failure*) instead the rate of failure is constant because it's more rare that a break occurs once the first period of operation is exceeded. Finally, the increase in the third part of the graph is clearly due to the normal wear of the device, that over time will naturally tend to break.
Burn-In test aims to age a DUT to verify that it exceeds the infant mortality phase, thus decreasing the probability probability of having extrinsic defects. A device tested with this methodology when placed on the market is already in the central part of its life cycle.
The aging is achieved by stressing the device with levels of voltage, current and temperature that go beyond its normal operating range. The correspondence between Burn-In and aging time is determined by a mathematical relationships [13] that usually are strictly dependent on the voltage, current and temperature with which the DUT is stimulated.

2.5 System Level Test (SLT)

Unlike previous methodologies, SLT involves a functional test of a device. The aim of the last is to verify the functioning of a device in an environment that simulates its final context of use. For example, if the DUT is an automotive microcontroller, by applying functional patterns [7], interaction with other vehicle devices can be verified: reading from sensors or peripherals, writing external flash memories, checking communication protocols, and so on.
The cost for this type of test is relatively low respect to a full pin test, but more expensive than a simple Burn-In test, that usually stimulates a low number of pins. Moreover the SLT can be easily updated by adding new functionalities [5]. A device before being placed on the market is tested several times and in different ways: from the production of silicon wafers, up to functional tests. Consequently, the test techniques described in this chapter can be performed in cascade. The duration of SLT is in the order of minutes [5], to optimize costs (in terms of time and money) it is possible to combine together different types of tests, for example the union of Burn-In and SLT [2]. Clearly to accomplish the last task, it is necessary to modify the test boards used for Burn-In, in order to integrate what is necessary for SLT.

Chapter 3

Tools and Software Environments

The following is an introduction of the hardware and software tools used to accomplish this thesis work. All listed information are freely available from the websites of the software providers. The SPC5-Studio development environment [27] will be described initially, and then the software PulseView [25]. Finally the target MCU [29] will be presented.

3.1 SPC5-Studio

SPC5-Studio is an Eclipse-based environment that allows to develop software applications in a simple and intuitive way for the *SPC5 32-bit microcontroller family* produced by STMicroelectronics. The software is free and downloadable directly from the website [27]. To start a new software project is sufficient to select a target MCU, or a development board among the many available (Figure 3.1). In addition, the environment integrates a series of ready-to-use examples. The software provides graphical interfaces to configure the target device, eventually the environment generates the folder hierarchy and the corresponding source code based on the configurations chosen by the developer. The source code includes the HAL, a set of software libraries that abstract the hardware level to provide the functions necessary for its use. All the settings are stored in an xml file (*configuration.xml*) in order to facilitate the export to other environments. Another useful file is the makefile, by editing it, is possible to set some compilation options and select the files to include in the compilation process. In addition SPC5-Studio is able to manage both C and C++ source files simultaneously.
The bare metal development mode is not the only one, indeed the source code of the Real Time

Operating System (RTOS) FreeRTOS can be included in a project. Another notable feature is the compatibility with the MISRA C standard. The latter is a set of guidelines for the development of software written in ANSI C whose purpose is to ensure greater security, reliability and portability of the code. In automotive and other fields where it's necessary to minimize any errors resulting from incorrect writing of the software, this standard is widely required.

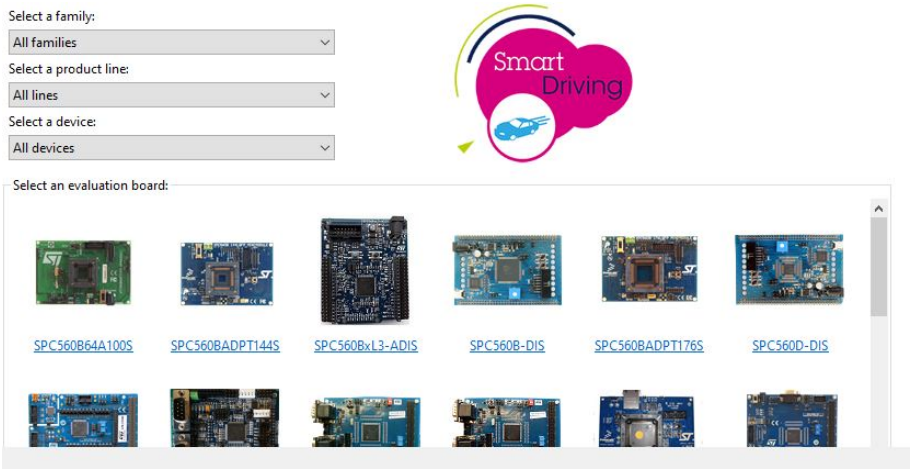

Figure 3.1: SPC5-Studio target device selector.

Some of the configurations that have been useful in the development of the applications covered by this thesis are now described. The *PinMap Editor* (Figure 3.2) graphically handles the setting of the GPIOs, according to the MCU target datasheet, it's possible to set a GPIO in input, output or as an alternative function (SPI, GTM, UART and so on) and specify some electrical properties like pull-up or pull-down. In addition to the GPIOs, the environment gives the ability to enable, disable or configure (interrupt configuration, DMA mode, callbacks setting and so on) also the other peripherals. Through the corresponding window are managed the SPI bus settings: transmission frequency, polarity and phase. In the following chapters the Generic Timer Module (GTM) peripheral will be the subject of an in-depth analysis, SPC5-Studio provides interfaces for interaction with this device as well. Even the clock tree can be set by graphical user interface, moreover the software checks the inserted values, avoiding incorrect configurations.

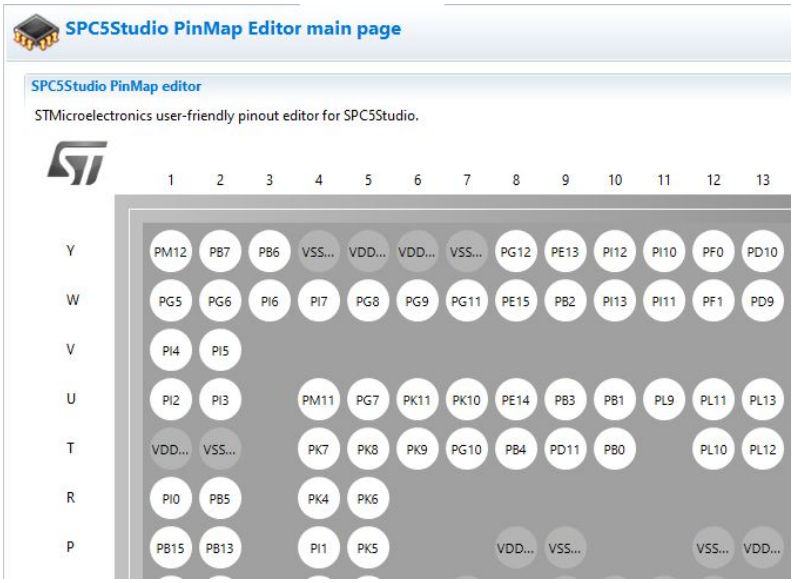

Figure 3.2: SPC5 Studio PinMap editor.

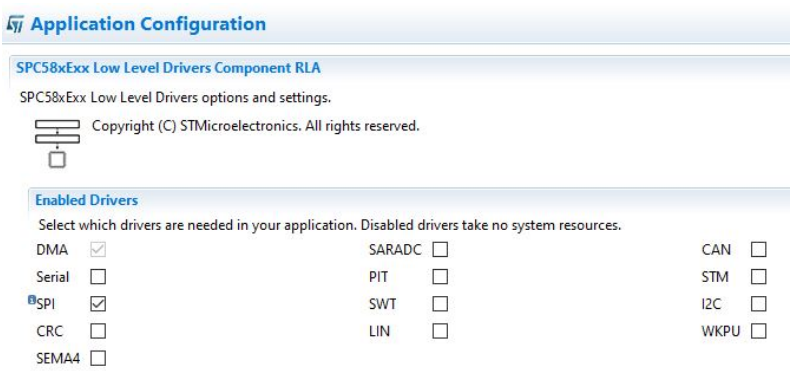

Figure 3.3: SPC5-Studio enable/disable peripherals.

In closing SPC5-Studio also integrates a direct access to the software UDE STK debugger [34]. The latter in *STARTER KIT* version is freely downloadable from the website. The debugger installed on the development machine is connected to the target MCU through a JTAG/USB dongle, the compiled object code is first flashed into the device's memory, and then executed and controlled step by step.

3.2 PulseView

PulseView is a software capable of analyzing the waveforms captured by a logic analyzer. In detail this device is connected to the signals to analyze and to a development machine where PulseView is installed. The software presents a simple interface in which is possible to view the logic analyzer input channels. From the main screen are selected the sampling frequency and the number of samples to capture, obviously these parameters should be chosen in such a way as to ensure the correct reconstruction of the signal, according to *Shannon Sampling Theorem*. The capture of a line can be activated manually, or by specifying a trigger in order to start the acquisition of samples at a specific moment. For example on the rising or falling edge of a clock signal. The capture will start only after the selected trigger is detected, and will end when all the samples are acquired. After the acquisition the captured waveform is displayed and then it's possible to analyze it with the tools provided by the application. These tools also include a decoder, useful to intuitively view data transmitted using the most common communication protocols. The Figure 3.4 is an example of capturing the SPI bus component signals, the trigger is set on the rising edge of the SCK signal. Moreover, the decoder encodes the transmitted bits in hexadecimal.

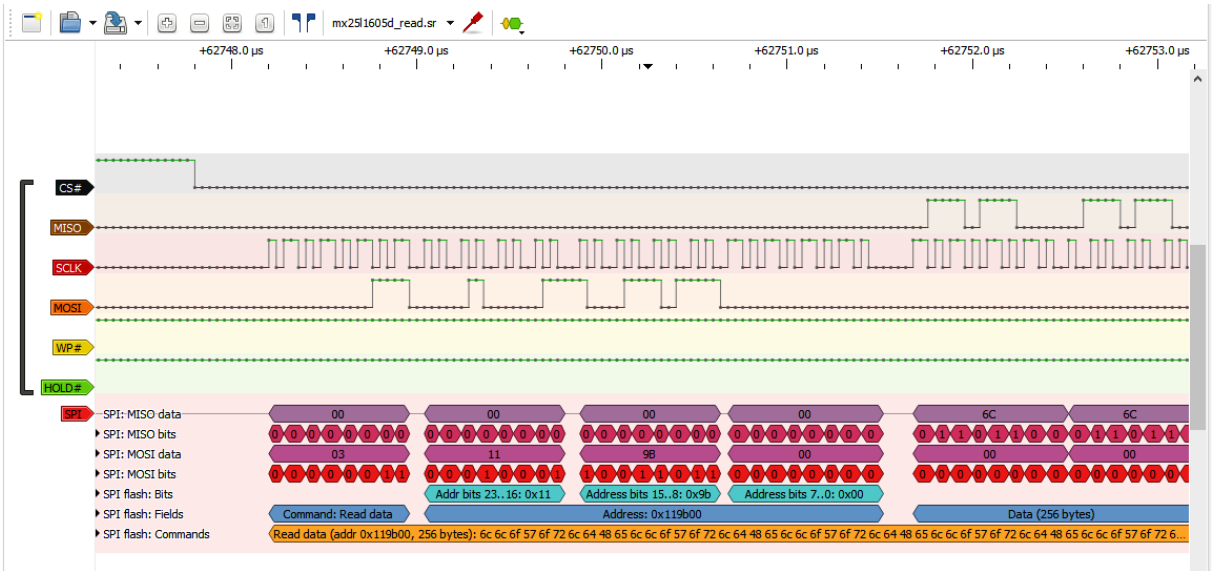

Figure 3.4: PulseView example capture SPI bus. Image from [25]

3.3 Target MCU

The target microcontroller for this thesis work is the SPC58NE84C3 [29] produced by STMicroelectronics. This device belongs to the *SPC58 family* of 32-bit automotive microcontrollers, it is mounted on the evaluation board [28] [30] (Figure 3.5). This last allows the developer to interact with all the peripherals of the MCU, providing all the connectivity needed.
Below some of the main device features:

- 32-bit PowerPC Architecture multi-core (three main CPUs);
- 6576 KB flash memory;
- 768 KB SRAM;
- GTM-343 device;
- 10 deserial SPI interfaces;
- Two Ethernet controller 10/100 Mbps (IEEE 802.3-2008 standard compliant).

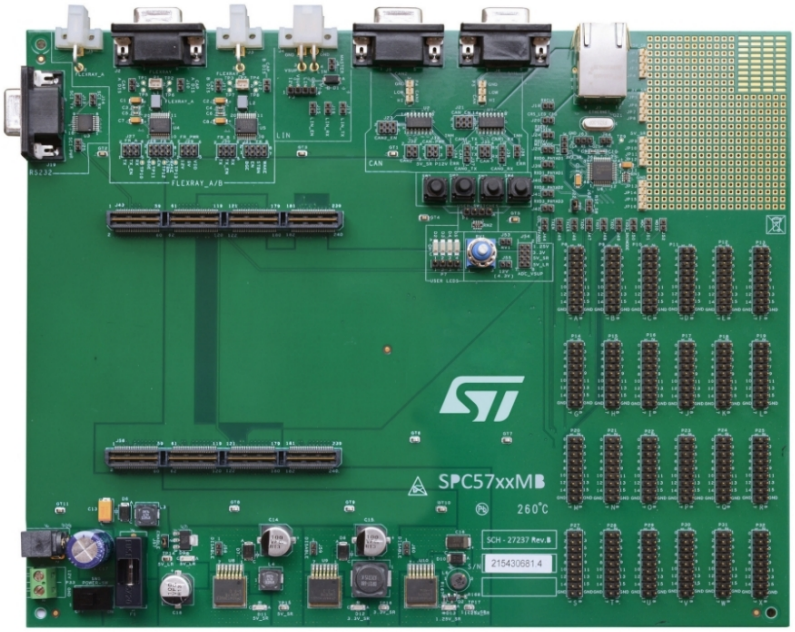

Figure 3.5: Development board. Image from [28]

Chapter 4

Vectorial Mode and Pseudorandom ATPG Pattern

The following paragraphs discuss two different software implementations, to be integrated into complex test environments based on FreeRTOS. Obviously in accordance with the objective of this thesis, particular attention will be paid to the resources of the supervisor MCU. The first of these techniques, designed by Ing. Francesco Peri (STMicroelectronics), refers to parallel control of a GPIO port in order to achieve a vector communication with a DUT. Each GPIO encodes one of the lines needed for the testing activities, some of these are part of JTAG standard, or signals required by an ATPG transmission. In the second part of the chapter, reference will be made to a Pseudorandom Number Generator (PRNG) algorithm [9], used for generating ATPG patterns (arrays) based on pseudorandom numbers. In the context of this section the arrays generated by the algorithm will be referred as chunk.

4.1 Vector Transmission

A digital signal can be shaped as a sequence of binary values. For instance to generate the clock signal in the Figure 4.1(a) using a simple GPIO pin, it's sufficient to set in series the logical level of the peripheral first to 1 (high) and then to 0 (low). Obviously a crucial parameter is the value T, indeed the latter establishes how long the value of the GPIO is held high (or low). This time interval is basically a delay that the CPU waits before setting the following logical level.

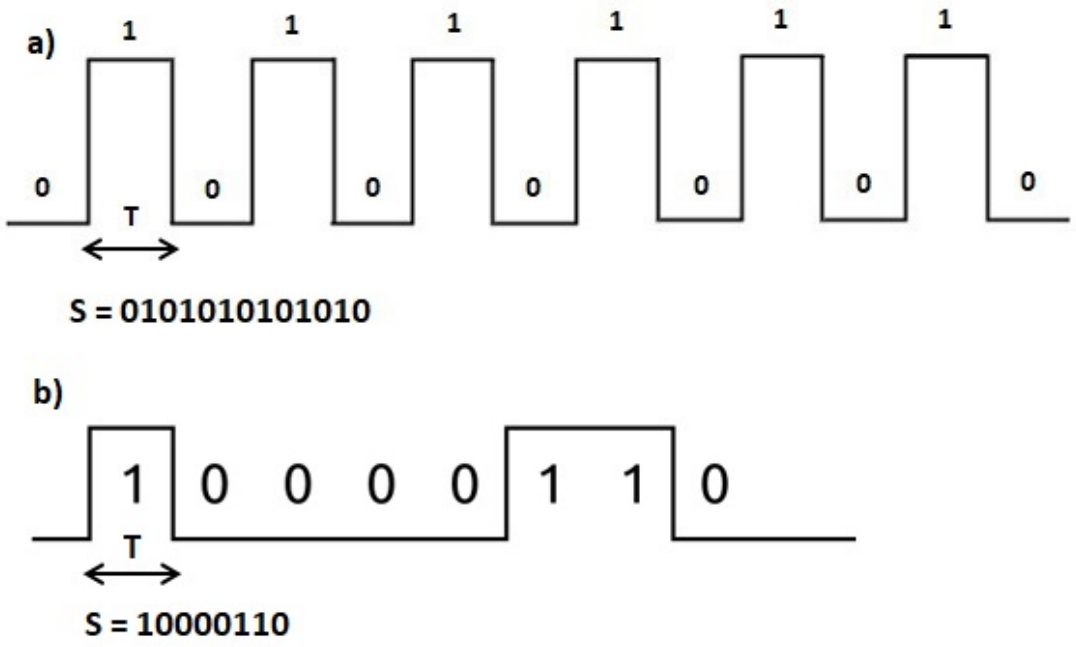

Figure 4.1: Digital signals as a sequence of binary values.

Clearly the same argument can be applied for any digital waveform, the only difference is the binary sequence used to encode it (the value of *S* in the Figures 4.1(a) and 4.1(b)). To generalize as described to multiple signals, a succession of *n-bit* words (Figure 4.2) encodes *n* different signals, and each of the *i-th* bit represents the binary value of a line in a certain instant. The previous figure can be seen as a simple SPI transmission generated in GPIO mode, of course the Figure 4.1(a) is the SCK line, while the Figure 4.1(b) the MOSI channel. In this case it would be necessary to control in parallel (taking into account the value of *T*) two distinct GPIOs, on the basis of as many sequences of binary values. The result is the same as that obtained with a real SPI bus.

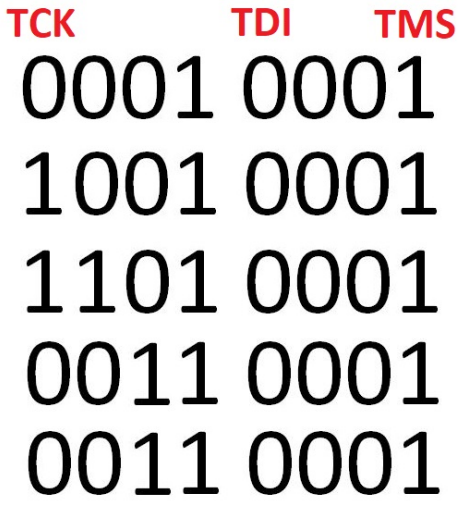

Figure 4.2: Time evolution of JTAG signals encoded in an 8-bit word.

For a JTAG or ATPG transmission several lines are required, their number depends strictly on the DUT to be tested. For example in the Figure 4.2 are depicted some of the lines included in the JTAG standard. The remaining bits may be used for other signals such as the TRST (which as described in paragraph 2.2 is not mandatory in the standard), or dependent on the particular DUT.

4.1.1 Arrays Architecture

Generally, the evolution of a signal can be stored in arrays whose size and type depend on how many lines are required by the transmission. For instance if the bus consists of eight separate signals, an array of *char* will suffice. Each of the element is thought in the way described in the previous paragraph.

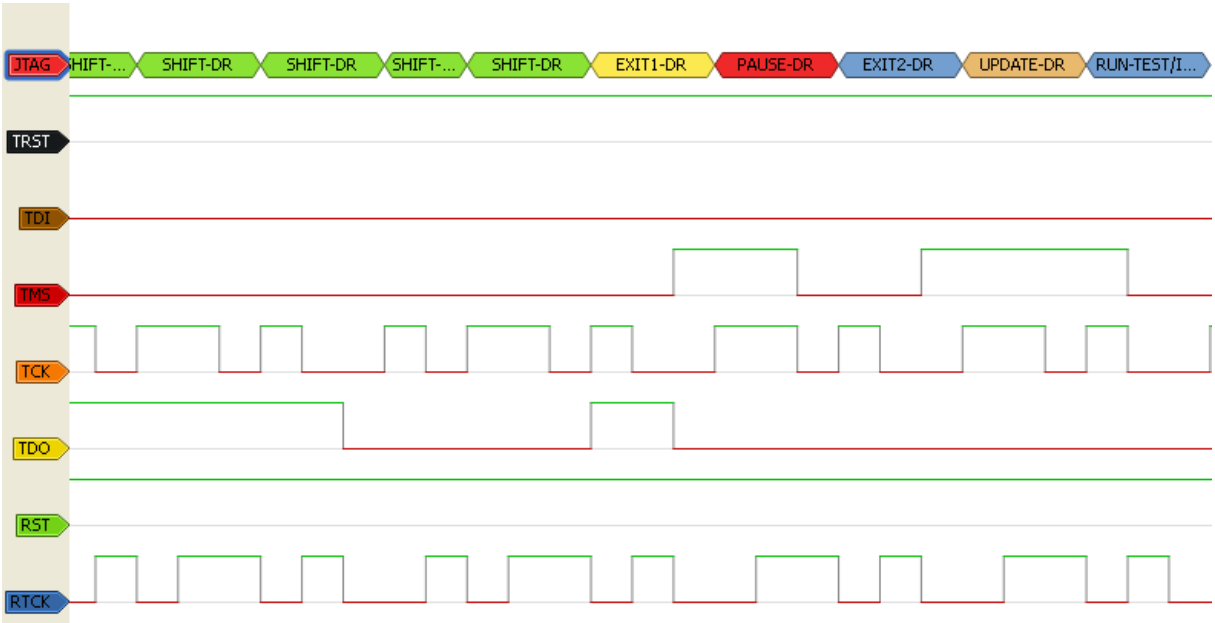

Figure 4.3: JTAG transmission of some stable signals.

Once the array is defined, it will be sufficient to scroll it along its length. The elements will be used to drive in parallel a set of GPIO pin belonging to the same port. The latter problem in the case of this work is immediately solved using the SPC5-Studio development environment. The software described in 3.1 includes among all functions that belong to the HAL libraries, the function *pal_writepad*. The previous receives as argument the GPIO port to drive and an integer whose bits encode the logical level of the pins belonging to the port. However using just one array, although an intuitive choice, is not the most efficient solution. The Figure 4.3 shows a set of signals that make up a JTAG transmission. Some lines including the TDO and TMS, do not vary frequently, they keep their value (high or low) for some time. Hence from the arrays point of view, a repetition of elements resulting in memory waste. The proposed solution defines a kind of "data compression", instead of using just one array, two of the same size are allocated: *Masktorepeat* and *Timestorepeat*. The former defines the waveform of the signals to

be transmitted, while the second stores the number of times that the corresponding element of the first array must be repeated. The following example clarifies the above:

$$MaskToRepeat = \{0x11, 0xA5, 0xBA...\} \tag{4.1}$$

$$TimesToRepeat = \{0x10, 0x05, 0x01...\} \tag{4.2}$$

$$MaskToRepeat[0] = 0x11 = (0001\ 0001)_2 \tag{4.3}$$

$$TimesToRepeat[0] = 0x10 = (16)_{10} \tag{4.4}$$

Since the array 4.1 contains 8-bit integers (*uint8_t*), it describes the evolution of sixteen distinct lines simultaneously. Starting from the first element (4.3), the binary values encoded are set in the chosen GPIO port (using the *pal_writepad* function) the number of times specified by the corresponding element of the second (4.4). The process continues until the array scrolling is finished. An extreme case is a signal that remains constantly low, with this method it will be enough just one 0, rather than a sequence.

4.1.2 Use Case: IDCODE Instruction

A simple use case of the above is the vector transmission of a JTAG instruction: IDCODE. As described in paragraph 2.2 the command allows to read an unique identification code from a device in *test mode*. The instruction is useful to understand if a DUT is properly configured, the ID contained in the response can be seen as a feedback. Following the model described in the previous paragraph, it's necessary to allocate the two *Masktorepeat* and *Timestorepeat* arrays. In addition to the command, the entire data sequence for *test mode* access is stored inside the array. Nevertheless in order to maintain the confidentiality, it will not be possible to show any real waveform, or give information about the nature of the arrays.

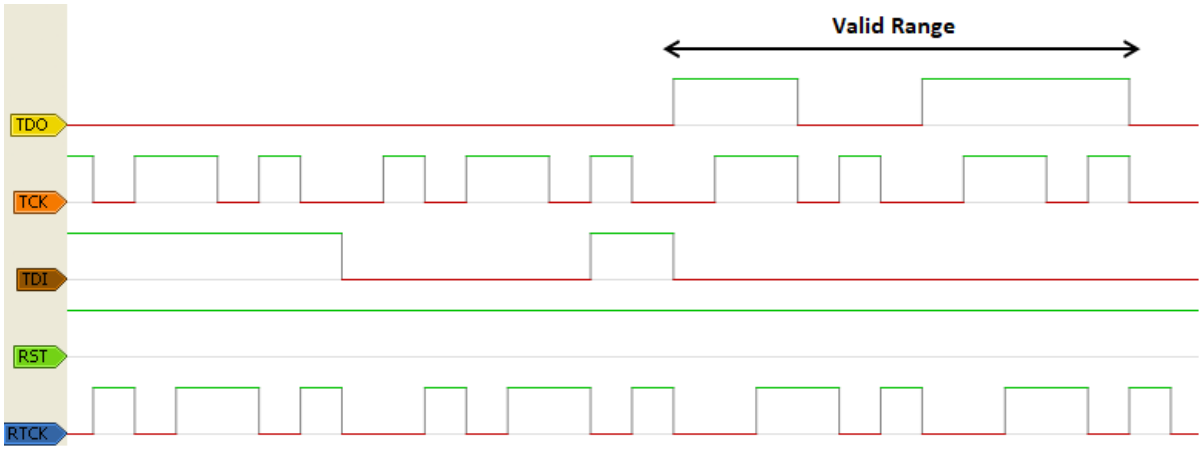

Figure 4.4: TDO valid range IDCODE instruction.

The proposed example is useful to understand how to use vector communication for reading a response from a device. The ID value is encoded in the TDO signal, so at a certain moment of time the line will contain the result of the IDCODE instruction. However as shown in the Figure 4.4, the TDO is valid only in a certain time range, so the reading must be done at the exact moment. The problem is easily solved by knowing the structure of the two arrays, indeed the capture is performed just in correspondence of the elements that contain the IDCODE instruction. The reading is done using another function provided by the development environment, in this case the *pal_readpad*. This function returns the level of a GPIO pin (in this case the one connected to the TDO line) passed as argument. Finally it's sufficient to store each sampled bit inside a receiving buffer. Once received the presumed ID and verified its correctness, the device is considered as properly configured.

4.2 ATPG Pseudorandom Generator Algorithm

In the following is described the integration of a Pseudorandom Number Generator (PRNG) algorithm, used for generating ATPG patterns, within a simple FreeRTOS environment. The patterns are array of pseudorandom numbers, their size can be considerable (even higher than available RAM), therefore the main memory occupation is a critical problem. One solution is to store them in flash memory, it's sufficient to declare the type as *static*. Nevertheless the application performance could degrade since the flash is certainly slower. In the proposed solution the memory saving comes from the possibility of generating an entire pattern one chunk

at a time, thus storing in main memory just the current chunk. The PRNG algorithm described below is able to generate arrays (chunks) of specified size, then is possible to build a complete pattern by composing successive chunks.
As shown in the Figure 4.5 an ATPG test requires at least three lines: SI, SCK, SO, it consists of two phases:

1. Sending stimulation pattern to DUT on SI line at each pulse of the SCK;
2. Result check by analysing the device response on SO line.

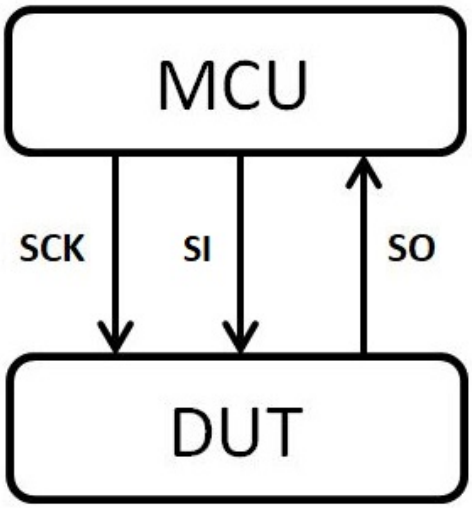

Figure 4.5: ATPG test connection schema.

Both previous are performed by the developed application, and will be described below. The application consists of two different FreeRTOS tasks: the first one simply occupies the CPU in order to emulate a real execution context; while the second handles the pattern transmission (1), or the SO control (2). Each task has the same priority, so the FreeRTOS scheduler will divide the processor resources equally. They are executed for a certain amount of time (slice), once expired the scheduler performing a context switching, will assign the CPU to the next task. Therefore, transmission or check operations can be interrupted at any time.

4.2.1 The PRNG Algorithm

The algorithm is widely described in [9], in this thesis work it will be considered as a black-box capable of generating array of pseudorandom numbers. Each element is a 16-bit integer (*uint16_t*), and the array size can be specified. The source code is written in C++ language, so as a preliminary operation it was necessary to integrate the algorithm within a complete C project. As discussed in 3.1, the development environment (properly configured) is able to perform a mixed compilation C and C++. Moreover each class method has been associated with a wrapping function such as the following, in order to transform them into simple C functions.

extern "C" void Wrap_Function(Class object) { object.method(); }

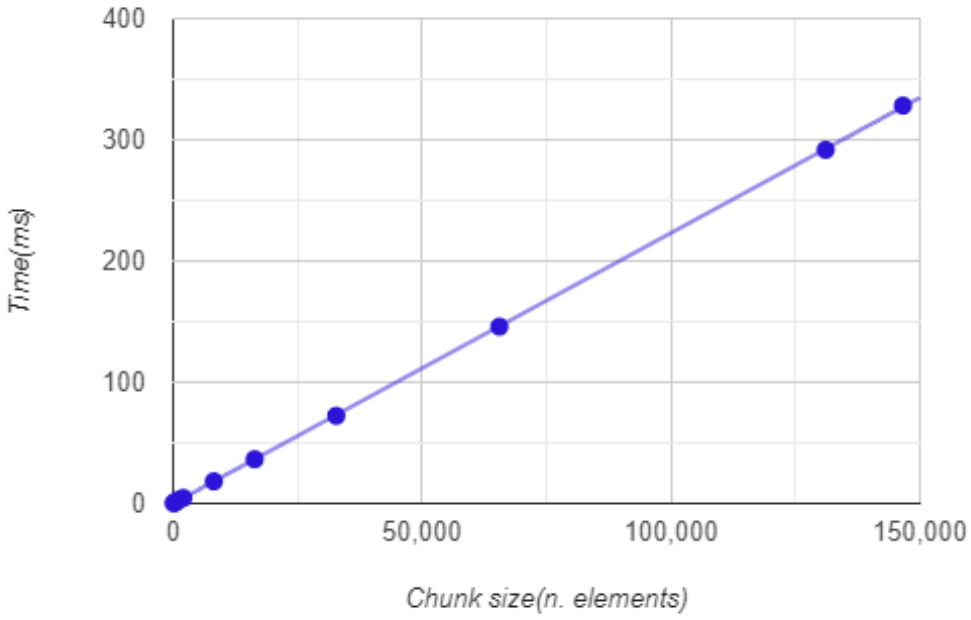

Figure 4.6: Chunk Generation Time.

In short the algorithm is based on polynomial arithmetic, a 32-bit integer represents a polynomial. The generation process starts from a seed so-called *initial state*, that is a device-dependent 32-bit number. At each iteration the current state is updated by applying a series of shifting operations

to achieve the next state. A pseudorandom number is obtained by considering the first 16 bits of the current state, then performing a cast. It's important to note that the PRNG has been modified in order to return the current state, in this way it's possible to generate an array sequentially. The algorithm guarantees a period (the amount after which the pseudorandom series is repeated) at least equal to the scan-chain length. Then there will be no repetitions, and each number of the sequence will be different from the previous one. Finally the Figure 4.6 shows the linear trend of the PRNG execution time, as the chunk size changes.

4.2.2 Scan-Chain Transmission

The first operation to perform is the pattern transmission to the DUT, in this case pseudo-random numbers. As explained in 2.3 two signals are required, SI and SCK, respectively data (ATPG pattern) and clock line. Clearly choosing a simple SPI bus is possible to use the MOSI line as SI and SCK as clock. Moreover the bus is configured to send 16-bit frames, exactly the size of a number generated by the PRNG.
The *send* function included in the APIs provided by the development environment has the following prototype.

void spi_lld_send(SPIx, txBuffer, size)

The first parameter specifies the chosen SPI peripheral from those provided by the supervisor MCU, the second and the third respectively the buffer to be sent, then a chunk of pseudorandom numbers, and the number of elements that compose it. Recall that a scan-chain is a set of flip-flops where the stimulation patterns are stored, in this context they will be considered synonyms. Once set the chunk size, it's possible to divide the entire scan-chain into a series of contiguous piece. Obviously the division may not be exact, indeed if the chunk size is not a scan-chain length divider, there will be one last incomplete chunk.

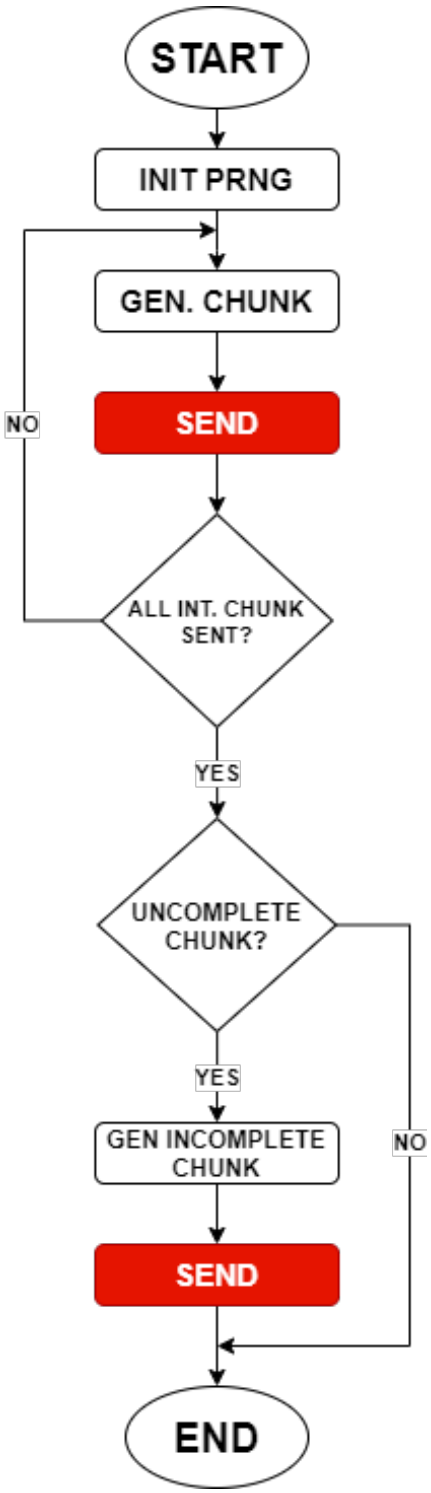

Figure 4.7: Sending the scan-chain: flow of operations.

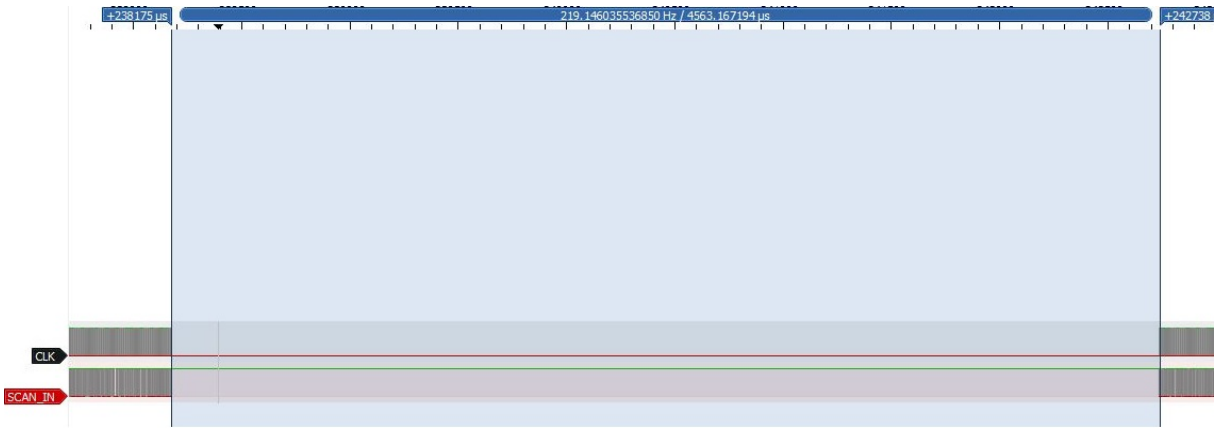

Figure 4.8: Transmission delay.

The PRNG is initialized with the *initial state*, then the algorithm is executed and all the generated chunks are sent to the DUT using the SPI bus. The process continues until all integer chunk have been sent, if present the incomplete is generated and sent. The latter sending is not included in the loop because the number of items to send is different. The flow of operations is described by the diagram 4.7.

This first approach has an obvious limit: since all previous operations are executed by a FreeRTOS task, as soon as the time slice of the task expires, the process will be interrupted. As a result, the transmission time increases and delays (Figure 4.8) may appear between the sending of one frame and the next. Also not considering the interruption caused by the scheduler, the sending operation is blocking (red boxes in the Figure 4.7). Hence before generating the next chunk, the transmission of the current one must be completed.

The following solution takes advantage of Direct Memory Access (DMA) system, in this way the *send* is no longer blocking, indeed the CPU can execute the PRNG to generate another chunk, while the DMA controller handles the transmission. Moreover when the task is removed from the processor by the scheduler, the sending continues since the operation is managed in DMA mode. The instructions remain the same described in the Figure 4.7, the difference is just the sending operation, and in addition the manual setting of the DMA controller. Each time the *send* is called, DMA registers (source, destination, data size) are configured, then it's essential to be sure that the transmission has been completed before calling the function again. Otherwise the overwriting of the registers will cause errors in the data.

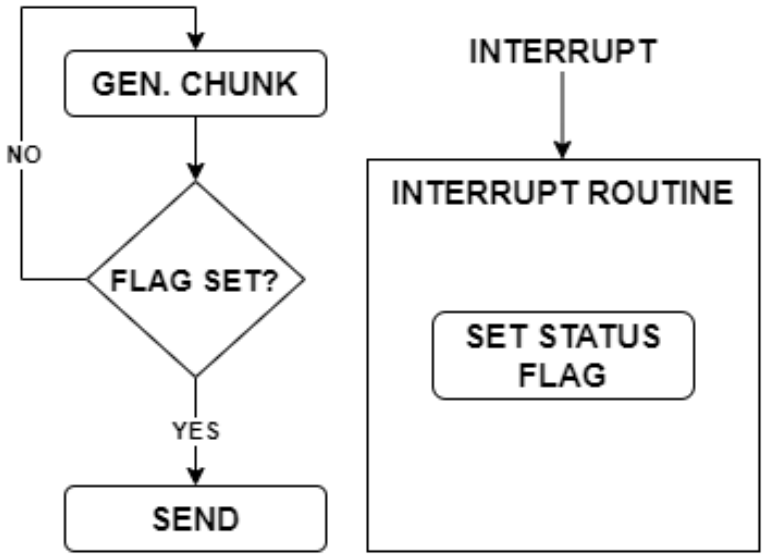

Figure 4.9: DMA control operations.

In the Figure 4.9 is depicted the control mechanism to ensure the respect of the previous constraint. When the DMA controller ends a chunk transmission, it raises an interrupt. Then the CPU is triggered and executes an interrupt routine that sets a status flag. In the meantime the task continues to generate chunks, only if the flag is set another chunk is sent. This mechanism assures that DMA registers are not overwritten, and although new chunks are available, they are transmitted only when DMA is free.

4.2.3 Scan-Out Control

Once the ATPG pattern has been sent and stored in the scan-chain, the next step is to check test results by analyzing the device's response to stimulation, the latter is encoded in the SO line. For this test it's expected that the SO signal is identical to SI, so the result is correct only if the two are the same waveform. Remember that a scan-chain is basically a shift-register, so at each clock pulse of the SCK signal the DUT will emit a bit on the SO line, corresponding to a bit of the ATPG pattern. Since as discussed in the previous paragraph, a stimulation is a series of pseudorandom numbers built chunk by chunk, also in this case the operation will be executed sequentially, generating again all the chunks. In view of this, the test control proceeds as follows:

1. Sixteen clock pulses are emitted on the SCK line, using SI of the SPI bus (MOSI) configured to transmit 16-bit frames;

2. On the SO line the DUT sends the bits corresponding to a 16-bit integer, a psedudorandom number of the stimulation pattern;

3. Previous operations are repeated until a sequence of numbers corresponding to the chunk size is obtained;

4. The PRNG algorithm generates a chunk, and a function checks that coincides with the one just received from the DUT. If the operation is successful, the process resumes from step (1) until all the integer chunks are checked. Otherwise it's interrupted returning an error.

Finally if an incomplete chunk is present, the instructions described from step (1) to (4) are performed again. It's important to recall that for the latter possible chunk, the operation must be differentiated. Indeed, the call to the *send* function will be modified to transmit a buffer of different size.
Also in this case the problem of blocking sending is repeated. Indeed the SPI transmission occupies the CPU, and it is interrupted at each context switching from the scheduler. In addition, the generation of new chunks cannot be performed until the current transmission has been completed. Thus the same solution is proposed as in the previous case: DMA mode. The advantage is even greater, since unlike the previous case, the tasks that the CPU may perform while the DMA controller manages the transmission are two: chunk validation and PRNG execution.

4.2.4 Results

This section shows the application performances during the transmission of an ATPG pattern generated according to the model described above. The paragraph will refer to both DMA and not-DMA versions, in order to make clear the differences between the two modes of sending. The pattern size (that is also the scan-chain length) is arbitrary and does not refer to any existing device. However the number is not random, it was chosen looking for a compromise between the duration of sending and the accuracy of the logic analyzer. The last is used to verify the correctness of signals that make up the transmission.

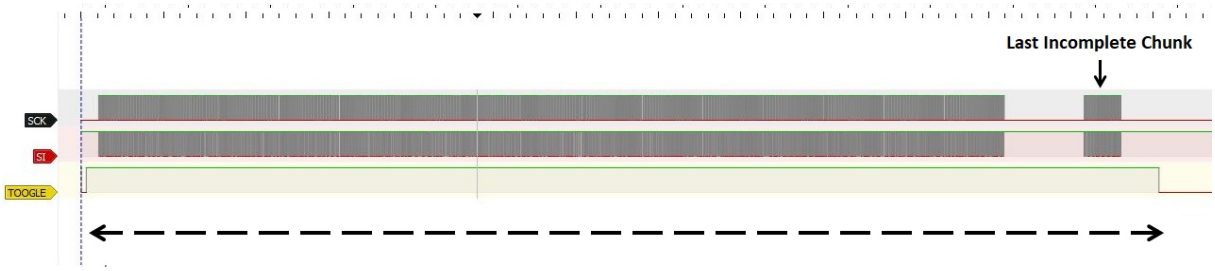

Figure 4.10: How to measure the transmission time.

Obviously in a real case the patterns size is certainly greater, however for the purposes of the experiment it is a significant value. The same arguments also apply to the choice of the SPI bus sending frequency, in this case the rate of 2.5 Mhz has been selected. A value too large (by virtue of *Shannon's Theorem*) would require a sampling frequency and a number of samples unsustainable by the logic analyzer and the development machine.
Before commenting on the results, it's important to specify some details about the measurements: to ensure greater accuracy, rather than using the library function *osGetMicroSecond* (that returns the number of *us* since System Timer starts), it's preferable to perform the measurements manually, as described in the Figure 4.10. Before to begin the transmission, a GPIO pin (*TOGGLE* line in the figure) is set to high (1), while at the end the same pin is reset to low (1). Thus the duration of sending can be obtained by measuring the time interval in which the *TOGGLE* line has an high logical level.

For the analysis the following parameters will be considered:

- Chunk size;
- Memory;
- RAM occupation;
- *bss*;
- Sending Time;
- Sending Time DMA.

The first four parameters contain information about the memory occupation of the chunks. *Chunk size* specifies the number of elements that make up a chunk, then the array length. The next one instead indicates the corresponding value in kilobyte, while the term *bss* refers to the main memory size of the compiled object code. *RAM occupation* is the percentage of the target microcontroller's RAM occupied, considering it all available. Finally the last two parameters respectively specify the transmission time using DMA and not.

Chunk size (n. elements)	*bss*	Memory (kB)	RAM occupation (%)
128	39878	0.25	0,03%
256	40134	0.5	0,07%
512	40646	1	0,13%
1024	41662	2	0,26%
2048	43710	4	0,52%

Table 4.1: Memory occupation of the chunks.

Clearly as can be seen from the Table 4.1, memory occupation in terms of all the parameters increases together with the size of the chunks. The values considered start from 0.25kB to 4kB (typical values for this kind of application), the latter are chosen by referring to the RAM of the target device, which in this case does not even reach 1MB. Furthermore as described above, development is not *bare-metal*, some of memory is consumed by the operating system, thus the memory cannot be occupied only for storing chunks.

To better underline the advantage obtained with the proposed solution, consider that the entire pattern would occupy a size of 9.77MB, much more than the 768kB available on the target device. Indeed, trying to compile the code by setting the chunk size equal to that of the entire pattern, the following compilation error would be obtained (Figure 4.11)

```
build/out.elf section `.bss' will not fit in region `ram'
region `ram' overflowed by 9657700 bytes
```

Figure 4.11: Compilation error: RAM overflowed.

Chunk size (n. elements)	Sending Time (mm:ss:cc)	Sending Time DMA (mm:ss:cc)
128	01:09:11	00:58:33
256	01:09:07	00:58:17
512	01:09:04	00:58:11
1024	01:09:05	00:58:11
2048	01:09:01	00:58:03

Table 4.2: Sending time: locking and DMA mode.

In conclusion, Table 4.2 shows the performance obtained by the application in terms of transmission time. Whether using DMA or not, the chunk size does not affect the duration. Instead the time difference between the two solutions is evident, in the case of DMA the performance improves on average by 15%. This is intuitive since the CPU rather than managing the transmission generates other chunks reducing delays and therefore the total time. Also during the context switches executed by the FreeRTOS scheduler, the sending continues. Instead the blocking version that does not use DMA shows worse performance, indeed no new chunk can be generated along with the transmission, and in addition it is interrupted at each context switch.

Chapter 5

Generic Timer Module (GTM)

In this chapter we will talk about the GTM peripheral. A detailed description is outside the scope of the thesis, just some important details for the application developed and described in the next chapter will be introduced. All contents below refer to the technical documentation of BOSCH [18], NXP Semiconductors [20] and STMicroelectronics [16]. The Generic Timer Module (GTM) is an IP module designed by BOSCH, and included in some automotive microcontrollers in order to minimize the effort of the CPU in the related task execution. A working example is the engine carburetion, an application in which real time constraints are required. The peripheral is composed of several modules, there are more than one unit for each module, hence more replicas of the same module. The latter can be combined to build complex applications: use the TIM to capture input signals; generate PWM signals with TOM or ATOM; advanced signal elaboration inside the MCS and so on. In the figure 5.1 is depicted the logical structure of the GTM. The CPU interacts with the peripheral via Generic Bus Interface (AEI). The clock of the GTM is configurable by setting the MCU clock tree.

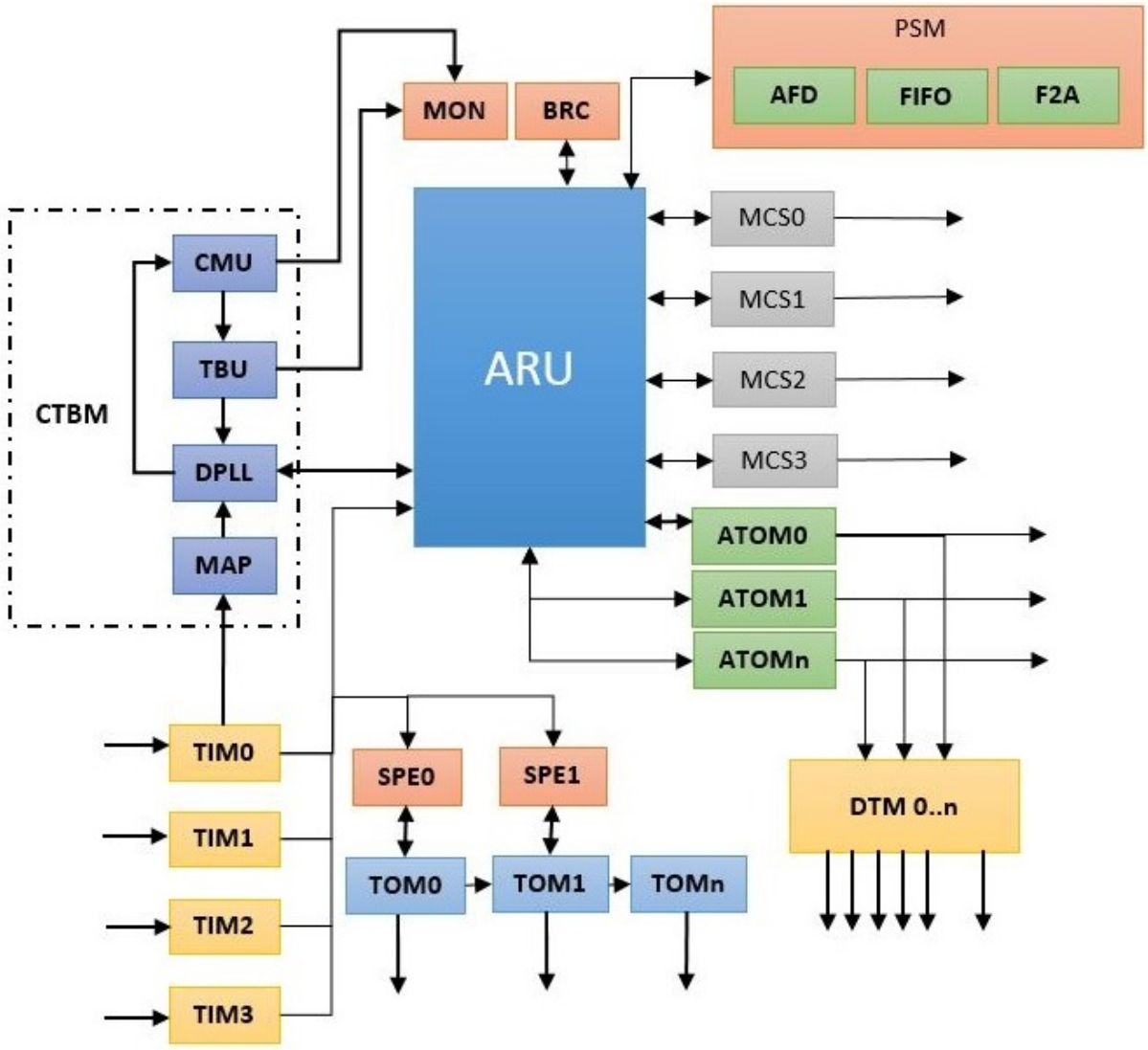

Figure 5.1: GTM logical structure overview: component modules. Image from [27]

5.1 Advanced Routing Unit (ARU)

One of the most important features of the peripheral is provided by the ARU, it allows to connect some of the modules. The ARU is basically a router that transfer data from a data source to a data destination: for each module connected to the ARU there are several sources and destinations. The data exchanged consists of a word of 53 bits, in the figure 5.2 is shown its composition.

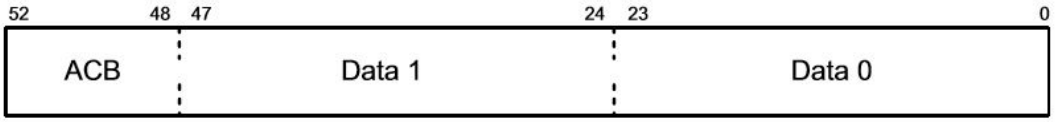

Figure 5.2: ARU data word. Image from [16]

Bits 0 to 23 and bits 24 to 47 contain the application data, for example the duty cycle and the period of a PWM signal. The last ARU Conrol Bits (ACB) are used for control purpose, the meaning of this bits is strictly dependent on the module that uses it. For instance in the ATOM channels this bits can be used to set the shift direction of the data. With just the ARU is not possible to connect more than two modules, in this case the BRC must be used to transmit the same data word to more modules, and therefore build a broadcast transmission. As mentioned above, there are several data destinations and data sources for each module connected to the ARU. In the Figure 5.3 is depicted an example of this connections.

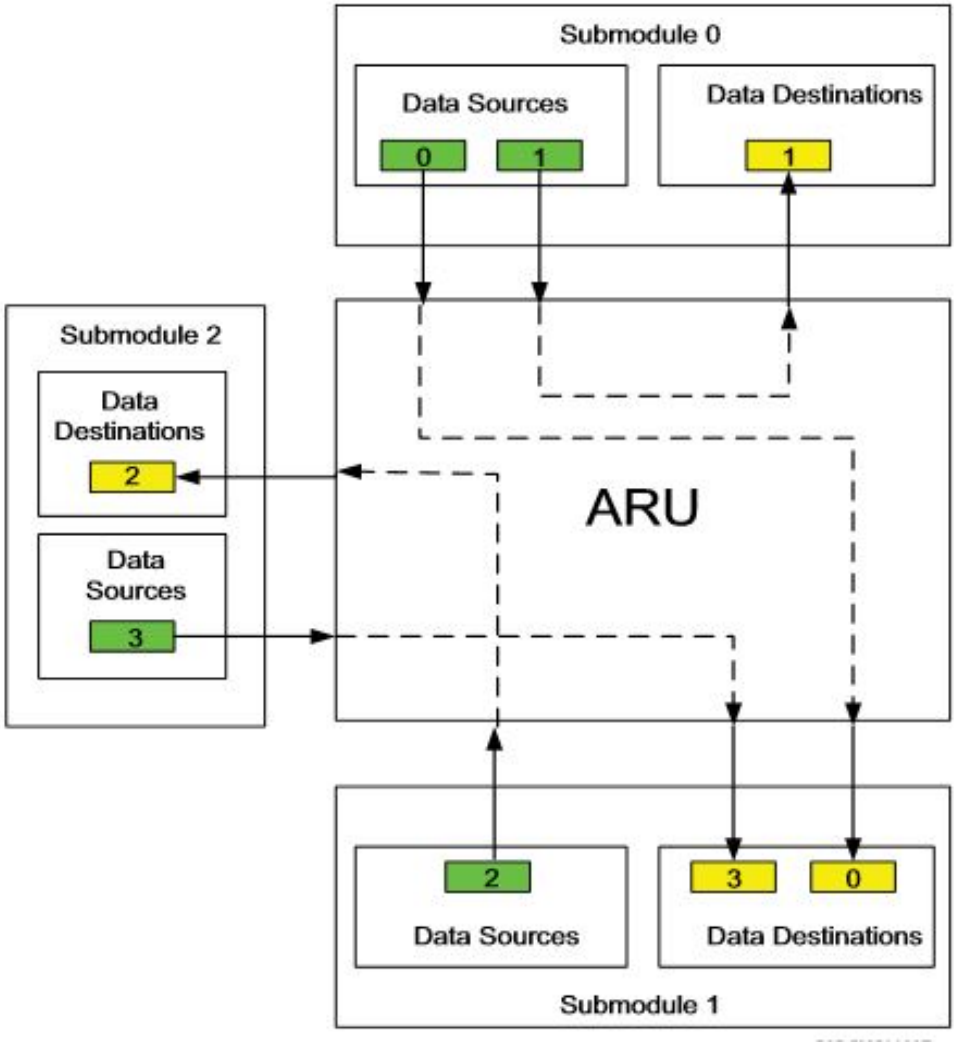

Figure 5.3: Interconnection of data sources and destinations through the ARU. Image from [16]

Every data source or data destination have an unique address used to identify them. The destination awaits data from the source, the reading is destructive and for this reason is not possible to connect more destinations to the same source without the BRC.
The ARU routing mechanism is the following:

1. The destinations are sequentially interrogated by the ARU in *round-robin* order;
2. When a data is available from the source, and the destination requires it, the ARU transfers it and communicate the delivery to both. Then, the source indicates this data as consumed.

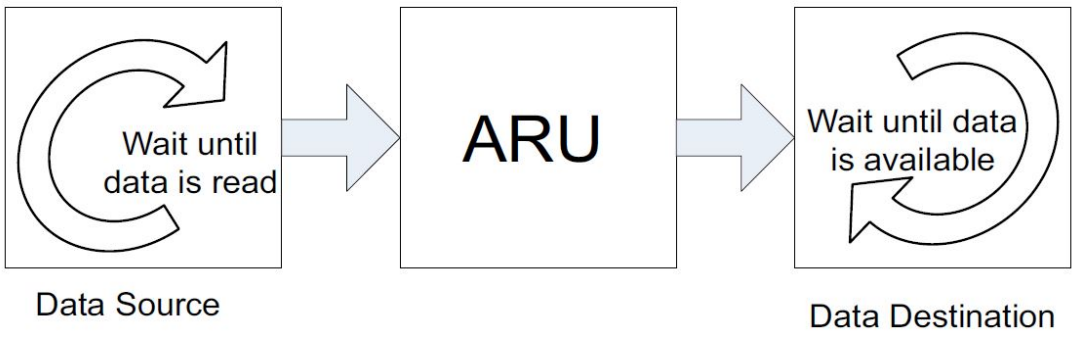

Figure 5.4: ARU blocking mechanism. Image from [16]

A connection between source and destination is "blocked", as depicted in the Figure 5.4 if a destination requests data from a source, it has to wait until the data are available. However the module containing the data destination can perform other operations while waiting. Besides the data source can send new data to the ARU, only if the old data is read (consumed) by the destination. Since there are multiple modules that may require ARU services at the same time, a mechanism to arbitrate them turns out to be necessary. The ARU performs a deterministic *round-robin* scheduling with a fixed Round Trip Time (RTT) (specified by the version of the GTM), so if the ARU is busy the time awaited from the last of two adjacent requests is always equal to an RTT.

5.2 Parameter Storage Module (PSM)

The FIFO sub-module is part of the PSM module, which also includes other two sub-modules: FIFO to ARU Unit (F2A) and the AEI to FIFO Data Interface (AFD), respectively interfaces to the ARU and the AEI bus, this latter is the bus used by the CPU to communicate with the GTM.

Figure 5.5: PSM module composition.

Basically the FIFO is a RAM storage, each unit is divided into eight channels. It's possible to configure start and end address of each channel, operation modes (*normal* or *ring buffer*) and other controls such as read protections or fill level control. The sub-module is large 1K words, and the word size is 29 bit. This number is not random, it is exactly the dimension of a part of an ARU word (23 bit), plus the five control bits ACB. The two operation modes are now described:

1. In the *normal* operation mode the channel behaves like a common FIFO: the first word to be read (and then destroyed) is the last inserted;

2. The *ring buffer* operation is useful when is required a continuous data stream trough the ARU, the use of the F2A is mandatory in this case.

As mentioned above the FIFO sub-module is a single RAM, it's mapped into the memory address space of the CPU, and hence accessible from the AEI bus. In addition there are the other two interfaces F2A and AFD, for this reason a priority mechanism to arbitrate this three interfaces is provided by the PSM. The dimension of an ARU word is 53 bits but the FIFO memory word is only 29 bit, so when using the F2A interface there are configurable several modes to handle the data transfer: transmit the entire ARU word (53 bits), or just a sub-part of it (the first, or the second 24 bits beyond the ACB). As an example in the Figure 5.6 an entire ARU word is sent: the second part of the word (bits 24-47) is stored in the first location of the FIFO, while in the following location is stored the first part. The ACB bits (48-52) are duplicated in both word.

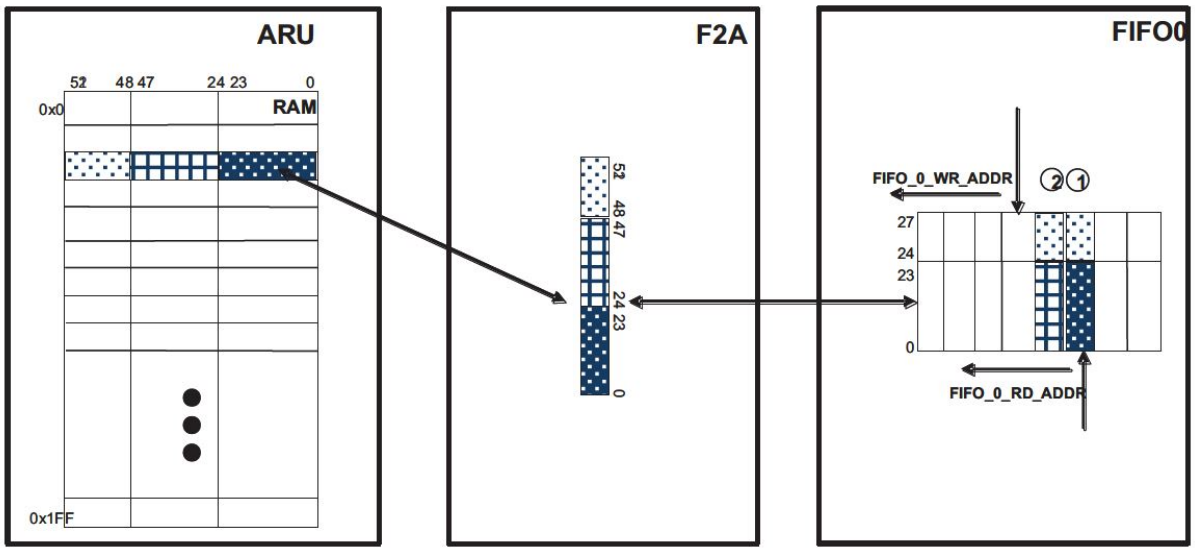

Figure 5.6: PSM, ARU example of data transmission. Image from [16]

5.3 Timer Input Module (TIM)

The TIM module is used to capture external signals, these last can be characterized in several manners. As well as the most of the GTM modules also the TIM has a numbers of channels. The signals to capture are connected to different channels of the module, each of the previous includes an FLT sub-module to perform filtering operations on signals entering the channel. For example the prefiltering is useful to detect and eliminate glithces that may appear within a signal. A glitch is essentially a short and sudden pulsation that may appear during transmission, as all forms of noise if not managed properly might corrupt the information content.
As mentioned above this module provides different measurement modes, however just the TIM Bit Compression Mode (TBCM) mode will be described, because it is chosen for the development of the application described in the next chapter. A channel in TBCM mode (available only for the TIM channel 0) captures all input signals to the module at a certain moment of time, determined by the choice of a trigger. Since the TIM is divided into m channels when the trigger is detected (instant sampling) all m channels are sampled, the logical level of the connected lines (0 if the signal is low, 1 if the signal is high) determines a word of m bit. The latter is stored in the least significant part of the GPR1 register each time a trigger is detected. The CNTS register specifies (by setting the bits appropriately to 1) the trigger that activates channel sampling: from

bit 0 to *m*-1 the rising edge of the corresponding *i-th* channel, the next *m* bits instead select the falling edge. Then for instance if the CNTS register contains the value 0x01, is chosen as trigger the rising edge of signal connected to channel 0, that could be a clock line. When the register contains multiple bits set to 1 the trigger is formed by the logical OR of all events.
Of course the module can also take advantage of the routing service offered by the ARU, so by setting its configuration bit to 1 (ARU-BIT) the content of the GPR1 register (along with other data) is sent to the specified module through the ARU. The Figure 5.7 summarizes all previous information.

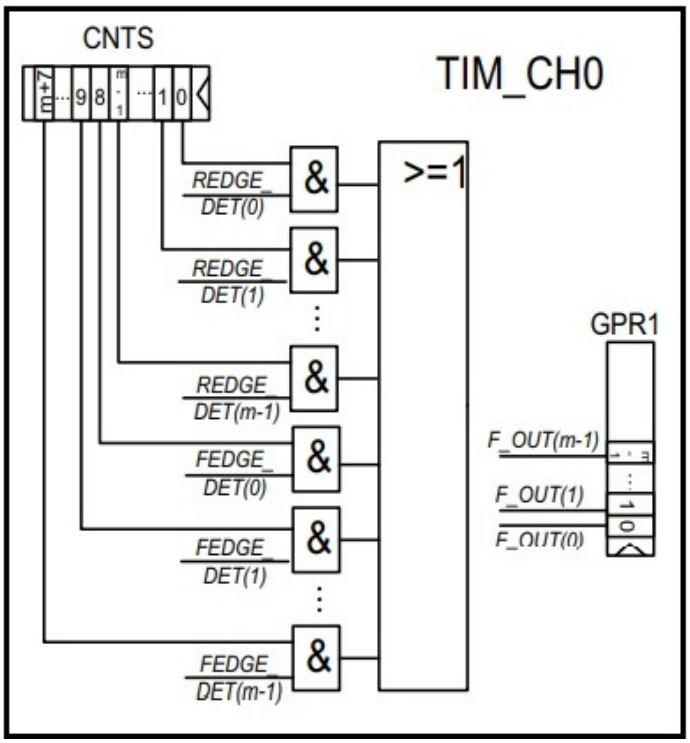

Figure 5.7: TIM Channel 0: TBCM mode.

5.4 Multi Channel Sequencer (MCS)

The MCS is basically a general purpose CPU with an ISA based on *PowerPC* architecture. The operating frequency is configurable by modifying the clock tree of the MCU. The module has a proper RAM that is mapped into the memory address space of the CPU. In this memory are stored data and instructions (generated by an integrated compiler in the development environment) for the MCS, but it can used as well to communicate with the CPU, or with another task (channel) of the same module. Just like the other modules of the GTM, also the MCS is divided into channels, and each of them is considered a separate processor "task". This last has its own instructions to execute, when more than one task is active they have to compete for the module's resources. Two scheduling modes are available for channels: *Round-Robin* scheduling or *Accelerated Scheduling Mode*. A single channel includes several register, obviously there are the Instruction Register and the Program Counter, in addition are available eight general purpose 24 bit registers, from R0 to R7. Just the trigger registers (STRG and CTRG) are shared between the channels, the latter are used to coordinate them. Finally the ISA can be divided into four categories:

- Data transfer instructions;
- ARU instructions;
- Arithmetic logic instructions;
- Control flow instructions;

The format of these instruction is "literal" or "double operand", the first operates with a single 24 bits register and a literal of the same dimension, while the second includes two registers as operands.

5.5 Clock Management Unit (CMU)

Some of the modules that make up the GTM need clock sources to work. For example, within the ATOM module clock signals determine the shifting speed in SOMS mode. The CMU provides a clock source for all modules that require one. The module is divided into the following three sub-modules, each of which handles clock signals in different modes:

- Configurable Clock Generation (CFGU);
- Fixed Clock Generation Unit (FXU);
- External Clock Generation Unit (EGU).

The CFGU generates eight different clock source, these are properly configurable to obtain the target frequency. Unlike the previous, the FXU produces four non-configurable clocks available only for the TOM and MON modules. It is defined "fixed" because is possible to set for each of the four sources, one of these values as dividing factors: 20, 24, 28, 212, 216. For this reason the clocks will have a fixed values. The last sub-module is used to produce clocks not for internal modules, indeed these signals are mapped outside the peripheral, in this case just tree output clocks lines are avaiable.

5.6 Aru-Connected Timer Output Module (ATOM)

To generate output signals the GTM provides two different modules: ATOM and TOM. The advantage of the former over the latter is the possibility to obtain the data flow necessary to characterise the outputs through the ARU. By connecting the ATOM to modules such as MCS or PSM, complex signals can be generated automatically and independently, thus saving CPU resources. The module is divided into channels, the type of output signals from each channel depends on the way it is configured. Among the most useful operating modes surely there is the Signal Output Mode PWM (SOMP), through the previous, a channel is able to emit PWM signals, simply by writing in the appropriate configuration registers the required period and duty cycle values. However, as in the case of the TIM module, only one operation mode will be described, the Signal Output Mode Serial (SOMS), as the one chosen in this work. An ATOM channel in SOMS mode behaves like a shift-register, at each pulse of the clock signal chosen as a reference, the channel emits a bit by shifting the register that contains the data to be sent. The

transmission rate (or shifting) depends on the reference signal, for each channel a clock source must be specified among those available in the CMU module. In addition to speed, shifting order and direction can also be configured. The number of bits to be transmitted is stored in the CM0 register, while the CM1 register contains the value. Each of the two previous registers is connected to a shadow register, SR0 and SR1 respectively. So the CM0 and CM1 registers are not written directly but will be loaded automatically after writing the corresponding shadow register. The transmission is interrupted when the CCU0 register that increases during shifting reaches the value stored in CM0. Clearly the advantage of the ATOM module compared to the TOM lies in the possibility of using the ARU, therefore through the latter the previous registers are modified independently without involving the CPU. In this case an ARU word contains in the least significant part (from bits 0 to bits 23) the amount of bits to be shifted that is the value of the CM0 register, while the most significant part (from bits 24 to bits 48) the data to be sent and then the contents of the CM1 register. Finally, ACB bits encode sending properties such as the shifting direction of the register.

Chapter 6

GTM Communication Interface

The purpose of this chapter is to describe an application of the GTM peripheral, in order to implement a different way of communicating between a supervisor microcontroller and a DUT. The main advantage is the reduction of the CPU workload, for instance the latter could perform other operations while the GTM transmits ATPG patterns or JTAG signals. Since the previous techniques require at least a data and a clock line, the communication model to be implemented is similar to an SPI bus. Therefore, two software models developed by BOSCH [17] and NXP Semiconductors [19] were taken as reference in the development of the following application. The aim of the latter is to emulate an SPI bus using the GTM. All the peripheral modules chosen for the project are described in the previous chapter.

6.1 The Proposed Architecture

This application can be thought as an SPI protocol between a master and a slave device. The input and output part described below work monolithically, for example is possible to connect the MCU to a real SPI master device and this would work properly. The choice of configuration depends strictly on the type of test architecture to be realized. In order to optimally describe the project it's convenient to divide it into two distinct sections: input and output. The TIM and PSM were chosen for the input part, while for the output the modules used are ATOM, PSM, and MCS. For both sections the ARU will manage the communication between the modules inside the peripheral. The Tables 6.1 and 6.2 summarize the modules, channels and the respective modes of use.

MODULE	CHANNEL	MODE
TIM	CH0, CH1	TBCM
PSM	CH0	NORMAL OP.

Table 6.1: Input: modules and channels configurations.

MODULE	CHANNEL	MODE
ATOM	CH0, CH1	SOMS
PSM	CH1	NORMAL OP.
MCS	CH0, CH1	ROUND-ROBIN

Table 6.2: Output: modules and channels configurations.

6.1.1 Input Description

The module chosen to capture input signals is obviously the TIM. As described in the previous chapter, it has different operating modes, but for the purpose of this application the most suitable one is the TBCM. The selected TIM channels are CH0 and CH1, in the first channel is connected the clock line, while in the second the data. The trigger is set on the rising edge of the clock signal, so each time the event is detected the data line will be sampled. The captured bit (stored into the GPR1 register) will be forwarded to the first channel (CH0) of the FIFO sub-module by the ARU. Basically the latter is a buffer capable of storing the received data. It should be noted that this operation is performed every time the trigger is detected and therefore bit per bit captured. For example, if the clock frequency is 1kHz, every microsecond the ARU will send to the PSM module a bit of the transmission. Moreover each bit occupies one FIFO data word, indeed the ARU transfers an entire 29-bit word for each clock pulse. Once a fill threshold is reached (then a certain number of received bits) the PSM module will raise an interrupt, hence the CPU will empty the FIFO and assemble the bits in a data buffer. The threshold is chosen by looking for a compromise between the number of interruptions raised and the memory occupation. Obviously a value too small will require a frequent use of the CPU, while too large can cause the saturation of memory. The Figure 6.1 summarises what has been described by identifying five distinct phases:

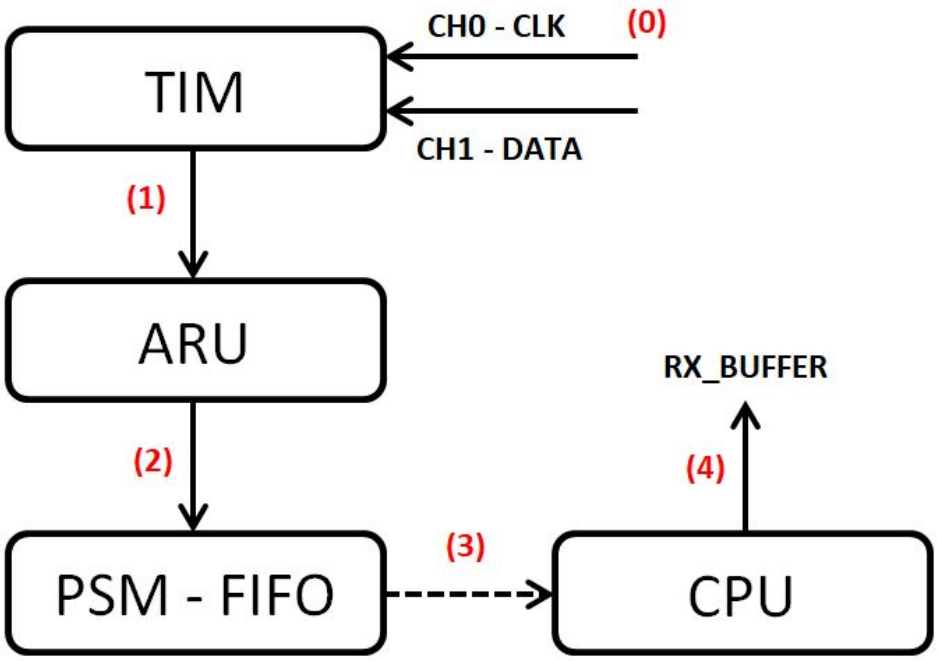

Figure 6.1: Input: flow of operations.

0. The TIM is activated on the rising edge of the clock signal (CLK) input to channel 0 (CH0), sampling the corresponding bit on the data line (CH1);

1. The sampled bit at the previous step is stored within a data word contained in the GPR1 register;

2. Through the ARU the content of the GPR1 registrer and then the current bit is forwarded to the FIFO sub-module;

3. When the FIFO fill level exceeds the threshold (sketched in figure) an interrupt that invokes the CPU is raised;

4. The CPU reads the FIFO words and composes a data buffer (*RX_BUFFER*).

The previous flow of operations is repeated until the transmission ends (there is no more data to receive), then the clock signal is interrupted.

6.1.2 Output Description

In the proposed architecture the ATOM generates the output signals, in particular the first channel (CH0) the data line, while the second (CH1) the clock. Both channels are configured in SOMS mode, hence each channel acts like a shift-register and able to emit (shift) bits at a rate determined by the target clock source selected from the CMU module.

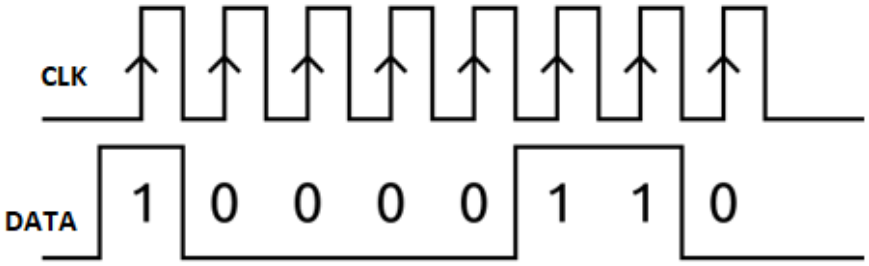

Figure 6.2: Correct waveforms.

Recall that when an ATOM channel is enabled in SOMS mode, the number of bits to shift and their value are stored in the CM0 and CM1 registers respectively. Even the clock line is treated like a data to be sent, indeed it can be encoded as a sequence of 0 and 1, the hexadecimal value 0x00AAAA. In order to achieve the correct behavior shown in the Figure 6.2, ensuring proper sampling of the data line, the channel CH0 is associated with an internal clock frequency twice that of CH1. This last consideration is crucial: if the data line was not stable at the time of sampling (on the rising edge of the clock), there could be errors in the transmission.
Also this part of the application makes use of the FIFO sub-module, but unlike the previous case the second channel (CH1) is selected to avoid data overlapping. The module stores in two successive words the content of the registers CM0 and CM1. These two memory locations are transmitted to the ATOM channels by the ARU.
The MCS module synchronizes all the modules involved, two channels are needed: the first (CH0) manages the transmission of the clock signal, while the second (CH1) the data line.
It's important to recall that an MCS channel is a task: a set of instructions written in a proper assembly language and executed by the module. From now on, to avoid confusion with ATOM channels, reference will be made to MCS channels as tasks: TASK0 and TASK1. The Figure 6.3 shows the overall process:

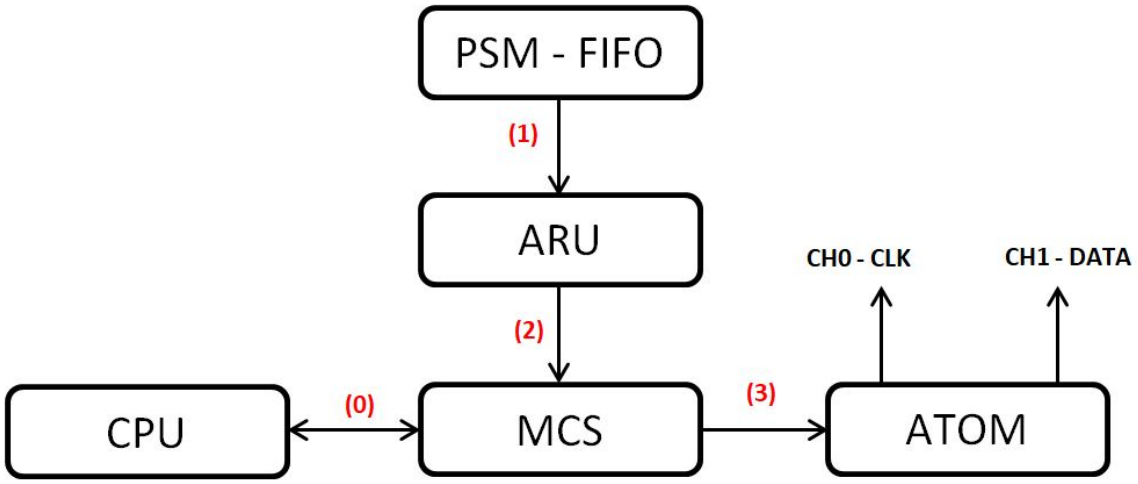

Figure 6.3: Output: flow of operations.

0. Before starting the transmission an handshake function synchronizes the CPU and the MCS. Then setting the value 0x01 inside the STRG, the CPU activates the TASK1. After it writes in the R1 register the number of bytes to be transmitted;

1. The TASK1 reads an ARU data word containing the data to be shifted and its size from the FIFO-submodule. These values are stored inside the R2 and R3 registers of the channel;

2. After the reading, the TASK1 enables (this time by writing 0x0 in the STRG register) the TASK0 which is responsible for managing the clock line;

3. At this point both tasks must use the ARU to write CM0 and CM1 registers of the ATOM channels CH0 and CH1. Once the previous are loaded, the shifting (transmission) will start automatically. The TASK0 that manages the clock line will write the registers with the values 0x00AAAA and 0x10, indicating respectively the clock pattern and the number of pulse. While the other the contents of the R2 and R3 registers received by the FIFO module at step 1).

The process is repeated until the value of the R1 register of the TASK1 (containing the number of bytes to be transmitted and decremented from time to time) is reset to zero.

6.2 Code Overview

The project is organized in a modular way, so the application can be configured by varying the choice of modules and channels. Please note that the GTM has multiple units of the same module, for example the choice of the first ATOM module is not mandatory and can be changed according to design requirements. The same considerations apply to all other settings.
The following files contain the functions needed to control and configure the peripheral: how to use a channel; loading the compiled code to be executed by the MCS; filling the FIFO with data; and so on. The overall structure of the code is summarised by the Figure 6.4.

- *ATOM_config.c*
- *PSM_config.c*
- *MCS_config.c*
- *TIM_config.c*

Each of the previous includes an header file containing the prototypes of the functions and the macros to specify the configurations. The functions to manage the transmission are included inside the file *GTM_lld.c*. The *struct GTM_config* stores the settings of a transmission, the parameters are similar to an SPI bus configuration:

- Clock frequency;
- Transmission modes: Least significant bit (LSB), Most significant bit (MSB);
- Phase, the logical level of clock line when the transmission is not running.

The previous *struct* is an attribute of the *GTM_driver struct*, the latter contains the following fields:

- A receiving buffer in which incoming data will be stored;
- The number of buffer elements;
- A buffer to save the data to be sent.

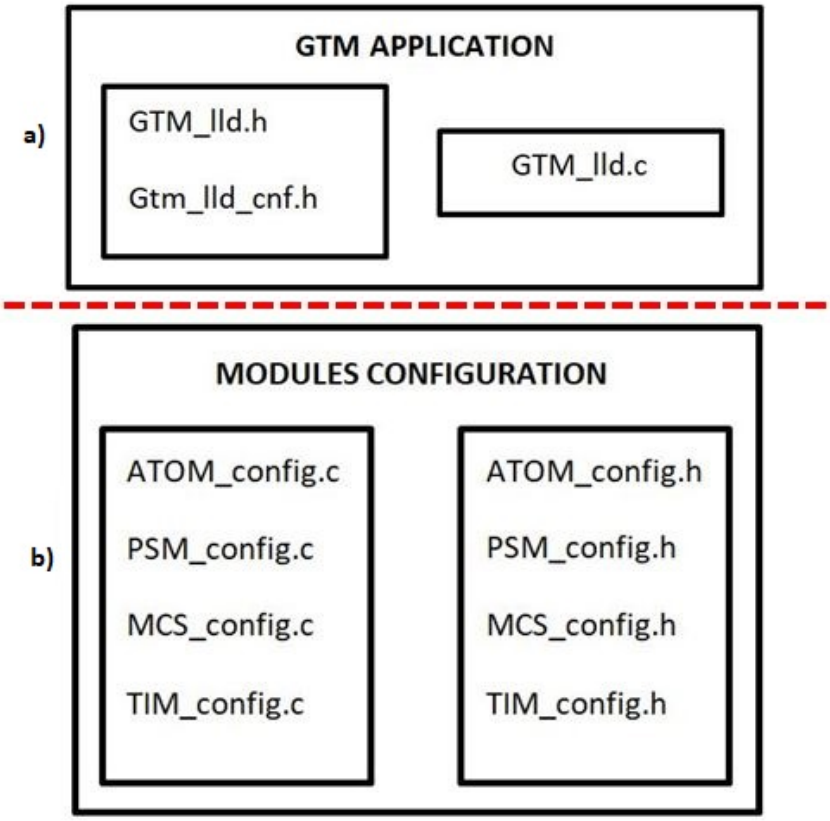

Figure 6.4: Code Structure.

As is shown in the Figure 6.4(a) the below functions, whose execution starts the entire transmission process described in the previous paragraphs, refer to those that directly manage the modules of the device.

1. *GTM_start*: configure GTM modules based on a parameter passed as argument;
2. *GTM_transmit*: start the transmission by sending the content of the *txBuffer*, at the same time the *rxBuffer* is filled with the received data (Figure 6.5);
3. *GTM_stop*: the call to this function causes the release of all the peripheral resources.

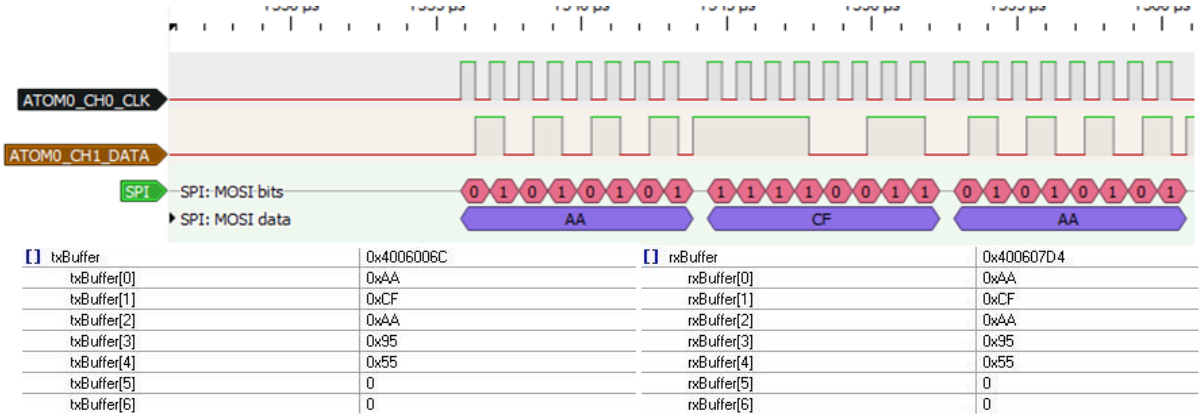

Figure 6.5: A correct transmission.

In the Figure 6.5 is depicted an example of execution of the previous ones. In this case the sending buffer transmitted is:

$$txBuffer = \{0xAA, 0xCF, 0xAA, 0x95, 0x55, 0x00, 0x00...\} \tag{6.1}$$

The ATOM channels from which the data and clock line are emitted, are directly connected to the TIM channels, so as to test the operation of both modules of the application. How the image shows, the data line (*ATOM_CH1_DATA*) correspond to the content of the *txBuffer*. In this case the transmission works correctly, the buffer where the received data are stored (*rxBuffer*) coincides with the *txBuffer*.

6.3 Transmission Limit

This architecture has a limit in the transmission rate. By considering just the case of the input the proof is immediate, however similar observations also apply to the output part and the application as a whole. The strength of the peripheral is the ARU, nevertheless it is also a bottleneck. The input capture is performed by the TIM module, each bit received during a transmission is stored inside the FIFO sub-module; then these last are directly connected by the ARU. As described in the previous chapter, when multiple sources require the ARU routing service, there is always a fixed time interval between two adjacent requests so-called RTT, and calculated as follows:

$$RTT = \frac{1}{C}\, SYS\_CLOCK \tag{6.2}$$

In the expression 6.2 the parameter C is a device-dependent constant, while the term *SYS_CLOCK* is the operating clock of the GTM (configurable by setting the MCU clock tree). The RTT value defines a limit in the data transmission rate:

$$RTT < Tc \tag{6.3}$$

If the values of Tc in 6.3 (the clock period of a signal to capture) is less than an RTT, then the ARU will not be able to forward the current bit to the FIFO before a subsequent bit has arrived, thus there will be an error in the transmission. A similar result was obtained by BOSCH in [17], this note also identifies a limit for the output part of the application:

$$RTT < 3\, Tc \tag{6.4}$$

Nevertheless the latter result is less significant, because the chosen architecture differs from that described in this thesis work.

6.4 Transmission issues

The proposed application has shown some problems. Initially will be discussed what happens when the transmission limit is exceeded, and subsequently an incorrect behavior in the generation of waveforms. The first experiment consists in sending an array of 8-bit integers (6.5), before at a frequency below the limit, and then at a higher rate.

$$txBuffer = \{0xAA, 0xAB, 0x12, 0xBB, 0xFF, 0xEF, 0xAE, 0xBA, 0xA6, 0xC8\} \tag{6.5}$$

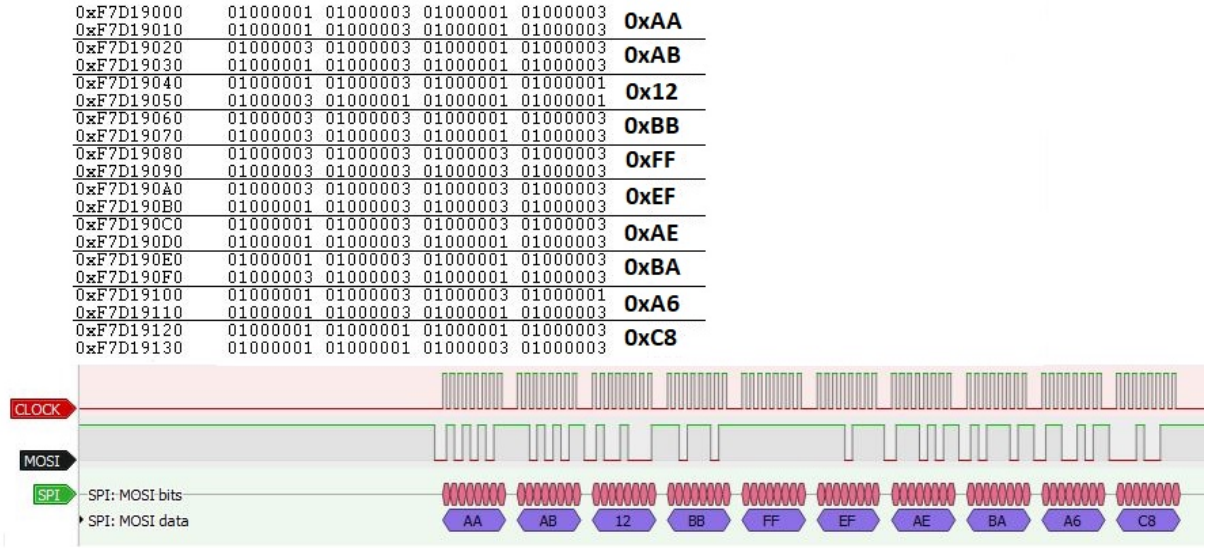

Figure 6.6: Correct input capture: transmission rate below the limit.

In the Figure 6.6 are depicted the waveform in which the buffer 6.5 is encoded and the FIFO sub-module memory dump, while the application is running. Please recall that every sampled bit by the TIM module is forwarded to the PSM module within an ARU word. Since each array element is an 8-bit integer, eight distinct data words are consumed, one for each sampled bit. For instance to recognize the value *txBuffer[0]* in the memory dump, consider the first set of locations, from the last (*0xF7D19010*) to the first (*0xF7D19000*): when the value ends with the digit three $(11)_2$ the bit is considered 1, otherwise 0. The explanation of this last statement can be easily understood by recalling what is described in the last paragraph (6.1.1). When the TIM module samples a bit of the transmission, the entire channel set is captured, not just the data line. Therefore the clock level, that is the first of the two bits must be ignored.

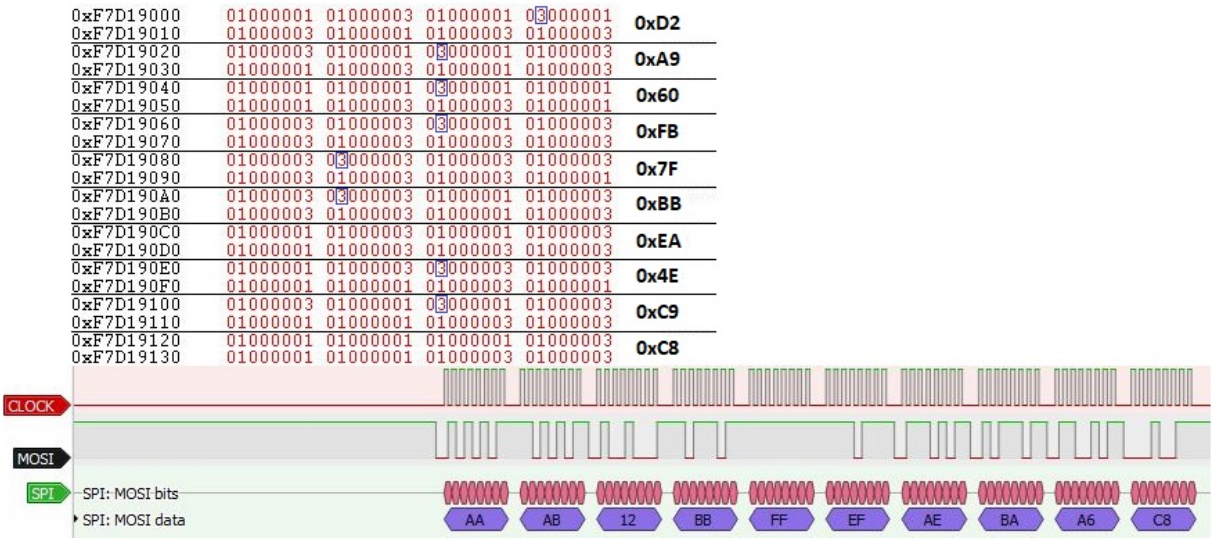

Figure 6.7: Input capture error: transmission rate over the limit.

How is shown in figure, in case of a correct transmission, the memory dump and the items of the sending buffer coincide. Differently in the following example (Figure 6.7) the memory dump is different. Indeed in this case the transmission rate is over the limit, then there will be errors in the capture. In the figure in blue are marked the bits that the ARU uses to report an overflow event (ARU Conrol Bits (ACB)): if the frequency is too high, the module fails to forward the currently captured bit to the FIFO, before the next one arrives. It's possible to note the overflow bit (*GPROFL*) also in the Figure 6.8, the latter shows the *C struct* that models the GPR1 register of the TIM module.

{} IRQ_NOTIFY	0xF7D0102C
R	9
I	9
{} B	0xF7D0102C
reserved_0	0
GLITCHDET	0
TODET	0
GPROFL	1
CNTOFL	0
ECNTOFL	0
NEWVAL	1

Figure 6.8: Input capture error: overflow bit.

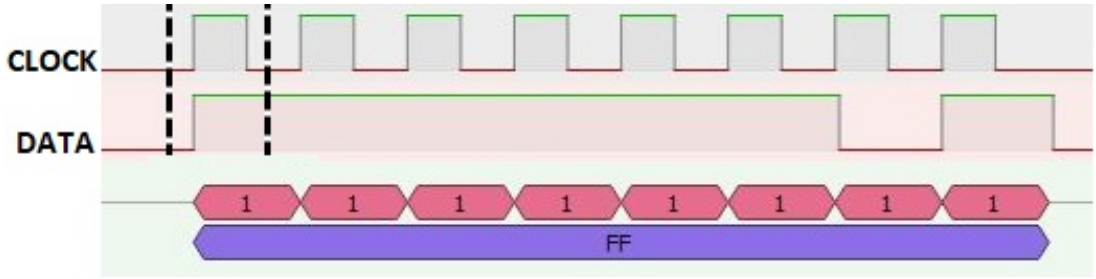

Figure 6.9: Incorrect waveforms: indeterminate sampling.

The other critical point (identified during the application test) is the capture of some incorrect waveform. The Figure 6.2 shows the proper behavior of the lines, in order to have correct data encoding. Remember that to obtain a right sampling it's necessary that the data line is already stable on the clock sampling edge. If they change simultaneously, the value of the captured bit is indeterminate. As can be seen from the vertical dashed lines in the Figure 6.9, the signals change level together, then since the data line is sampled on the rising edge of the clock, a random behavior is obtained.

Conclusions

The testing methodologies described in the State of the Art (2) are performed by means of complex and costly testers. They send to DUTs the test files containing a sequence of JTAG instructions to set internal configuration registers, in order to configure a DUT in the appropriate state to start a specific test. After the devices is configured, tests are performed via dedicated interfaces such as Scan In (SI) and Scan Out (SO).
The communication interface described in 4.1 derives directly from the possibility to convert the *stil* test file in simple arrays (this solution was developed Ing. Francesco Peri, STMicroelectronics). With this approach the complex *stil* files are translated into arrays easily managed by the supervisor MCU. This files are often complex and their understanding is not always easy. Nevertheless the solution is not problem free. First of all: the array size can be considerable; managing big arrays with the limited RAM resources of a microcontroller is not trivial. In second: the application described in 4.1 is based upon a blocking *delay function* necessary for signal generation; the problem can be solved by means of the DMA. Finally, the last consideration to be made is the possibility of adding the vectorial interface in a complex software testing environment such as the System Level Test (SLT). For instance the software module [24] can be integrated with the vectorial transmission mode, in addition to the SPI-based serial interface.

The integration of PRNG algorithm described in chapter five, is another example of how much an accurate software design (together with the hardware knowledge) becomes crucial to develop applications which are compatible with the resources of an MCU. The memory is managed by partition the information (ATPG pattern) in chunks of a manageable size. The DMA allows to perform transmissions more efficiently sparing the processor from being overloaded during the sending. Furthermore, the time savings obtained using DMA is not to be considered negligible, indeed during a test the patterns are sent several times, then the advantage is repeated at each sending.
The Generic Timer Module (GTM) was exploited to emulate a serial communication interface, since the peripheral is able to manage data transmission independently from the CPU. The computational resources of an MCU are always limited, especially when it has to run a testing software environment based for instance, on FreeRTOS. Thus the advantages of using the GTM are similar to those described for DMA in chapter five. Nevertheless considering the proposed architecture, the GTM scope strictly depends on the type of data to be transmitted. As an example in the case of ATPG pattern, since the scan-chain length reaches considerable dimensions, a low rate transmission could be limiting: the testing time would drastically increase.
The versatility is a clear advantage of the GTM, indeed by slightly varying the application (6), it's possible to connect the supervisor with multiple DUTs simultaneously. Would be required a more complex connection schema, to achieve the same result with an SPI based solution.
Finally, the thesis work proved that, if properly exploited, the resources and versatility of automotive microcontrollers can also be used in different context from their native applications. Besides, it has been shown that knowledge of hardware is essential in developing applications on systems so profoundly different from a general purpose CPU.

Acronyms

ABS Anti-lock Braking System. 9

ACB ARU Conrol Bits. 55, 58, 63, 74

ADC Analogical Digital Converter. 7, 8

AEI Generic Bus Interface. 53, 58

AFD AEI to FIFO Data Interface. 58

API Application Programming Interface. 8, 45

ARU Advanced Routing Unit. 55–58, 60, 62–68, 72–74

ASR Anti Slip Regulation. 9

ATOM Aru-Connected Timer Output Module. 53, 55, 62, 64, 67–69

ATPG Automatic Test Pattern Generation. 4, 18, 19, 27, 28, 36, 38, 41, 42, 45, 48, 50, 64, 77

BAS Brake Assist. 9

BIST Built-in Test. 19

BRC Broadcast Module. 55, 56

BSR Boundary Scan Register. 24, 25

CFGU Configurable Clock Generation. 62

CHPHA Clock Phase. 11

CHPOL Clock Polarity. 11

CMU Clock Management Unit. 62, 63, 67

CPU Central Processing Unit. 3, 5, 6, 8, 13, 14, 16, 42, 49, 53, 58, 61, 64–66, 68, 77, 84

DAC Digital to Analogical Converter. 8

DFT Design for Testability. 18

DMA Direct Memory Access. 13–16, 32, 47–52, 77

DSP Digital Signal Processor. 7

DUT Device Under Test. 3, 18, 19, 22, 24, 26–30, 36, 38, 40, 42, 47, 48, 64, 76, 77

ECU Engine Control Unit. 9

EGU External Clock Generation Unit. 62

F2A FIFO to ARU Unit. 58

FAN Fan-out Oriented. 28

FIFO Firt In First Out. 58, 65–69, 72

FXU Fixed Clock Generation Unit. 62

GPIO General Purpose Input/Output. 4, 7, 13, 32, 36, 37, 39, 41, 50

GTM Generic Timer Module. 4, 32, 35, 53, 57–59, 61, 62, 64, 69, 70, 72, 77

HAL Hardware Abstraction Layer. 8

I2C Inter Integrated Circuit. 7, 8

IC Integrated circuit. 18, 19, 27

IoT Internet of Things. 5, 6, 10

IP Intellectual Property. 53

ISA Instruction Set Architecture. 61

JTAG Joint Test Action Group. 4, 8, 18, 19, 22, 27, 36, 38–40, 64, 76

LSB Least significant bit. 69

MCS Multi Channel Sequencer. 53, 61, 62, 64, 67–69

MCU MicroController Unit. 3–10, 14, 16, 31–33, 35, 53, 61, 64, 72, 76, 77, 84

MISO Master Input Slave Output. 10

MISRA Motor Industry Software Reliability Association. 32

MOSI Master Output Slave Input. 10, 37, 45

MSB Most significant bit. 69

OS Operating System. 16, 17, 84

PODEM Path Oriented Decision Making. 28

PRNG Pseudorandom Number Generator. 4, 28, 36, 41, 42, 44, 45, 47, 49, 77

PSM Parameter Storage Module. 58, 62, 64, 65, 73

PWM Pulse-Width Modulation. 7, 8, 53, 55, 62

RTOS Real Time Operating System. 6, 8, 16, 17, 31, 32, 84

RTT Round Trip Time. 57, 72

SCK Serial Clock. 10, 27, 28, 34, 37, 42, 45, 48

SE Scan Enable. 27

SFF Scan Flip-Flop. 27, 28

SI Scan In. 27, 28, 42, 45, 48, 76

SLT System Level Test. 18, 30, 76

SO Scan Out. 27, 42, 48, 76

SOMP Signal Output Mode PWM. 62

SOMS Signal Output Mode Serial. 62, 67

SPI Serial Peripheral Interface. 4, 7, 8, 10, 12, 13, 32, 34, 35, 37, 45, 47, 49, 50, 64, 77

SS Slave Select. 10, 12

TAP Test Access Port. 22, 23, 25, 26

TBCM TIM Bit Compression Mode. 59, 65

TCK Test Clock. 22–24

TDI Test Data In. 22, 24, 25

TDO Test Data Out. 22, 24, 25, 39, 41

TIM Timer Input Module. 53, 59, 65, 66, 72, 73

TMS Test Mode Select. 22, 23, 39

TOM Timer Output Module. 53, 62

TRST Test Reset. 23, 38

UART Universal Asynchronous Receiver-Transmitter. 8, 32

USB Universal Serial Bus. 7, 8

VLSI Very Large Scale Integration. 19

List of Figures

List of Tables

Bibliography

[1] *32-bit Power Architecture MCU for Automotive General Purpose Applications - Chorus family*. STMicroelectronics. URL: https://www.st.com/en/automotive-microcontrollers/spc58ec80e5.html.

[2] F. Almeida, P. Bernardi, D. Calabrese, M. Restifo, M. S. Reorda, D. Appello, G. Pollaccia, V. Tancorre, R. Ugioli, and G. Zoppi. "Effective Screening of Automotive SoCs by Combining Burn-In and System Level Test". In: *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2019, pp. 1–6.

[3] T. A. Andrew S. Tanembaum. *Structured Computer Organization*. Ed. by Pearson. 6th ed.

[4] M. D. Applications. *MICROCONTROLLERS MADE EASY*. STMicroelectronics. URL: https://www.st.com/resource/en/application_note/cd00003980-microcontrollers-made-easy-stmicroelectronics.pdf.

[5] 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". In: *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2020, pp. 1–6.

[6] S. Bhunia and M. Tehranipoor. "Chapter 3 - System on Chip (SoC) Design and Test". In: *Hardware Security*. Ed. by S. Bhunia and M. Tehranipoor. Morgan Kaufmann, 2019, pp. 47–79. URL: http://www.sciencedirect.com/science/article/pii/B9780128124772000083.

[7] S. Biswas and B. Cory. "An Industrial Study of System-Level Test". In: *IEEE Design Test of Computers* 29.1 (2012), pp. 19–27.

[8] G. Bucci. *Architetture dei calcolatori elettronici*. Ed. by McGraw-Hill. 2001.

[9] P. Calao. "Development and evaluation of a low-cost Tester Architecture for Burn-In Test". Politecnico di Torino.

[10] H. Casier, P. Moens, and K. Appeltans. "Technology considerations for automotive [automotive electronics]". In: Oct. 2004, pp. 37–41.

[11] *Different Types of Microcontrollers are used in Automobile Applications*. URL: https://www.elprocus.com/different-microcontrollers-used-in-automobiles/.

[12] P. Draper. "Introduction to Reliability Engineering". In: Indico Cern. June 2015.

[13] *Early Life Failure Rate Calculation Procedure for Semiconductor Components. In Early Life Failure Rate Calculation Procedure for Semiconductor Components*. JEDEC Solid State Technology Association. 2007.

[14] S. Group. *IEEE Std 1149.1 (JTAG) Testability Primer*. Texas Instruments. 1997. URL: https://www.ti.com/lit/an/ssya002c/ssya002c.pdf.

[15] M. Grusin. *Serial Peripheral Interface (SPI)*. URL: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all.

[16] *GTM Reference manual*. STMicroelectronics. Sept. 2015.

[17] *GTM-IP Application Note AN016 GTM SPI application*. BOSCH. Feb. 2013. URL: https://www.bosch-semiconductors.com/media/ip_modules/pdf_2/gtm/gtm_ip_an016_mcs_spi_v03.pdf.

[18] *GTM-IP Generic Timer Module GTM-IP Specification*. BOSCH. Mar. 2016. URL: https://www.bosch-semiconductors.com/media/ip_modules/pdf_2/gtm/gtm-ip_specification_v3-1-5-1.pdf.

[19] I. Harris. *Generic Timer Module (GTM) Serial Peripheral Interface (SPI) Bus Emulation*. NXP Semiconductors. Apr. 2014. URL: https://www.nxp.com/docs/en/application-note/AN4864.pdf.

[20] I. Harris. *MPC57xxM Generic Timer Module (GTM) Quick Start Guide An introduction to the GTM as implemented on Freescale MCUs*. NXP Semiconductors. Apr. 2014. URL: https://www.nxp.com/docs/en/application-note/AN4351.pdf.

[21] *High-performance foundation line, ARM Cortex-M4 core with DSP and FPU, 512 Kbytes Flash, 180 MHz CPU, ART Accelerator, Dual QSPI*. STMicroelectronics. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32f446re.html.

[22] T. Jim. *The Two Percent Solution*. Dec. 2002. URL: https://www.embedded.com/electronics-blogs/significant-bits/4024488/The-Two-Percent-Solution.

[23] T. Kirkland and M. R. Mercer. "Algorithms for automatic test-pattern generation". In: *IEEE Design Test of Computers* 5.3 (1988), pp. 43–55.

[24] M. Lombardo. "Development of the Test Access Port Driver of a Test During Burn-In Coverage Enhancement System for Automotive SoCs". Università degli Studi di Palermo, July 2020.

[25] *PulseView User Manual*. Feb. 2020. URL: https://sigrok.org/doc/pulseview/unstable/manual.html.

[26] W. Rülling. "Design for Testability". In: *The Electronic Design Automation Handbook*. Ed. by D. Jansen. Boston, MA: Springer US, 2003, pp. 339–381. URL: https://doi.org/10.1007/978-0-387-73543-6_15.

[27] *SPC5-STUDIO Code Generator, Quick resources configurator and Eclipse development environment for SPC5 MCUs*. STMicroelectronics. URL: https://www.st.com/en/development-tools/spc5-studio.html.

[28] *SPC57XXMB Motherboard for SPC57 and SPC58 families of microcontrollers. Includes: universal power supply, documentation CD*. STMicroelectronics. URL: https://www.st.com/en/evaluation-tools/spc57xxmb.html.

[29] *SPC58NE84C3, 32-bit Power Architecture MCU for High Performance Applications*. STMicroelectronics. URL: https://www.st.com/en/automotive-microcontrollers/spc58ne84c3.html.

[30] *SPC58XXADPT292S Socketed mini module for SPC58 C, E, G and N lines in BGA292 package*. STMicroelectronics. URL: https://www.st.com/en/evaluation-tools/spc58xxadpt292s.html.

[31] *SPI Block Guide V04.01*. Motorola, Inc. July 2004. URL: https://www.nxp.com/files-static/microcontrollers/doc/ref_manual/S12SPIV4.pdf.

[32] *Technical Guide to JTAG*. URL: https://www.xjtag.com/about-jtag/jtag-a-technical-overview/.

[33] *The FreeRTOS™ Reference Manual API Functions and Configuration Options*. Amazon Web Services. URL: https://www.freertos.org/wp-content/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf.

[34] *Universal Debug Engine UDE and Microcontroller Debugger for AURIX, TriCore, Power Architecture, Cortex, Arm, XE166/XC2000, XScale, RH850, SH-2A, C166/ST10, STM32, Stellar, S32V234, S32*. PLS Development Tools. URL: https://www.pls-mc.com/products/universal-debug-engine/.

[35] *What Is Direct Memory Access (DMA) and How Does It Work?* URL: https://www.minitool.com/lib/direct-memory-access.html.

[36] *Why RTOS and WHAT is RTOS*. Amazon Web Services. URL: https://www.freertos.org/about-RTOS.html.

[37] Wikipedia. *JTAG — Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/JTAG.

[38] Wikipedia. *Serial Peripheral Interface — Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface.

[39] T. W. Williams. "Design for Testability". In: *Computer Design Aids for VLSI Circuits*. Ed. by P. Antognetti, D. O. Pederson, and H. de Man. Dordrecht: Springer Netherlands, 1984, pp. 359–416. URL: https://doi.org/10.1007/978-94-011-8006-1_8.

[40] X. Xiong, Y.-L. Wu, and W. Jone. "Reliability Model for MEMS Accelerometers". In: Dec. 2007.

[41] Xun Jiang, Xiaoxin Cui, and Dunshan Yu. "A JTAG-based configuration circuit applied in SerDes chip". In: *2011 9th IEEE International Conference on ASIC*. 2011, pp. 707–710.

[42] H. Yan, X. Feng, Y. Hu, and X. Tang. "Research on Chip Test Method for Improving Test Quality". In: *2019 IEEE 2nd International Conference on Electronics and Communication Engineering (ICECE)*. 2019, pp. 226–229.