UNIVERSITA' DEGLI STUDI DI PADOVA

Corso di Laurea Magistrale in Ingegneria Elettronica

a.a. 2015/2016

Release and Verification of an Operating System for Testing e-Flash on Microcontrollers for Automotive Applications based on Multicore Architecture

Studente: Claudio Menin
Matricola: 1065593
Relatore: Ph.D. Prof. Andrea Cester
Correlatore Aziendale: Ing. Angelo De Poli
Tutor Aziendale: Ph.D. Ing. Giambattista Carnevale

Padova, 11 aprile 2016

Abstract

The global automotive electronics industries are constantly growing as the cars produced contain an increasing number of electronic devices for active assistance to driving, safety controls, energy efficiency, passenger comfort and entertainment. Some future goals that we can mention are autonomous driving, gesture controls, 360° view parking assistance and so on. Safety is the keyword for these automotive systems and means to have electronic components high reliability. Infineon microcontroller division at Padua works to improve reliability and guarantee the quality of microcontroller flash memories. These last are actually tested by a firmware (single-core operating system Test-Application) called testware that aims to verify their proper functionalities through a long set of tests. The cost of these, on e-Flash memories, is high and highly dependent on the time employed in the tests. Due to the increase of number of electronic devices in cars, the size of flash memories increase more and more and then the company needs faster solutions to fully test these memories.

Table of Contents

Introduction

The microcontroller-team challenge is to realize a fast method for testing e-Flash memories. The initial idea was a single-core *Functional Test Operating System* (FTOS) that checks autonomously with various tests the memory reliability. The increase of memory banks that must be analyzed lead to the idea of designing a multi-core multi-task FTOS, to decrease as much as possible the flash memory test time with parallel analysis and maximize the operating system throughput. Thanks to the studies of several thesis students and the determination of the MC team of Infineon Development Center in Padua we arrived today in a first implementation of this multi-core multi-task operating system.

The aims of the thesis are the new operating system release and the verification of multi-core and multi-task modes.

This thesis is divided into seven sections: first chapter focuses on the introduction to the working context explaining why we adopted an operating system to test the microcontroller flash memory, the company project development approach and the tools used in the analysis. The second, the third and the fourth chapters give a brief overview of the flash memories architecture, the microcontroller architecture and the operating system structure. These informations are then used in the next chapters: the fifth chapter focuses on my contribution to the operating system release with the implementation and the verification of new synchronization functions and the solution to the code cloning problem; at the end of the chapter there is the multi-core verification. The sixth chapter explains the multi-task analysis performed and proposes a solution to fix an issue found in the multi-task verification. The last chapter recaps the obtained results and lists the project next steps.

1 Context Analysis, Company Methodologies and Tools

1.1 Infineon brief Presentation

The work presented in this brief document has been developed at the Infineon Technologies Development Center in Padua. Infineon Technologies is a German semiconductor manufacturer with headquarter in Munich founded in 1999 with more than 36,000 employees worldwide (as of Sep. 2015). Infineon focus areas are:

- Automotive (ATV)
- Industrial Power Control (IPC)
- Power Management & Multimarket (PMM)
- Chip Card & Security (CCS)

Infineon automotive market covers powertrain modules (engine and transmission control), comfort electronics (steering, shock absorbers, air conditioning etc.) and safety systems (ABS, airbags, ESP and so on). The product portfolio includes microcontrollers (MC), power semiconductors and sensors.

Padua Development Center activity in ATV MC sector is focused on embedded Flash testing in close collaboration with the partner sites of Villach, Munich and Singapore. The MC department is mainly split into a characterization team and a testware team. Product Engineering (PT) group works for characterization, validation and verification of device features and requirements. This group is involved in automotive microcontroller development activities, starting from first silicon analysis, till massive production support, trough customer validation, product qualification and testing support. Testware Engineers (TE) instead work on the development of software in order to speed up test execution and ideation of new test algorithms in order to increase test coverage.

1.2 FTOS

The quality and reliability required by the automotive industry are guaranteed by tight quality standards. The first approach used by the company to meet these standards for the test of e-Flash memories was the so called Build-In Self-Test (BIST), which is a main standard in many SoC modules. In this approach, a specific area on the dice is entirely dedicated to perform the desired tests on the target module. Anyway the BIST has several drawbacks, starting from silicon area occupation (consider that a flash module often occupies a huge percentage of the physical chip area), to the very low portability[1] of the designed test.

[1] Portability is the capability of being used on different systems. In this particular case, the systems are different e-Flash memories.

In 2002, Product and Test Engineers (PTE) chosen to use a different self-test for the SoC based on a software solution for embedded memories. This technique consists on using the DUT to test its memory itself, and this is possible only after a proper test of the necessary modules in the DUT (for instance, the CPUs, the RAM...). This is achieved by loading a dedicate test firmware (so called testware) into the already tested device's RAM. The testware is then executed by the processing units, which actually run the test. Testware fulfill quite well all the requirements of e-Flash testing, and the availability of DUT resources is powerful enough to support complex algorithms and to guarantee fast execution times. The PTE Padua team is owner of the library of e-Flash testware functions, and in 2007 the team developed a concept to standardize the testware layers [Figure 1], meeting given requirements to support different ATE machines and to have code easily portable among different DUT derivatives [1]. This was the birth of the Functional Test Operating System (FTOS).

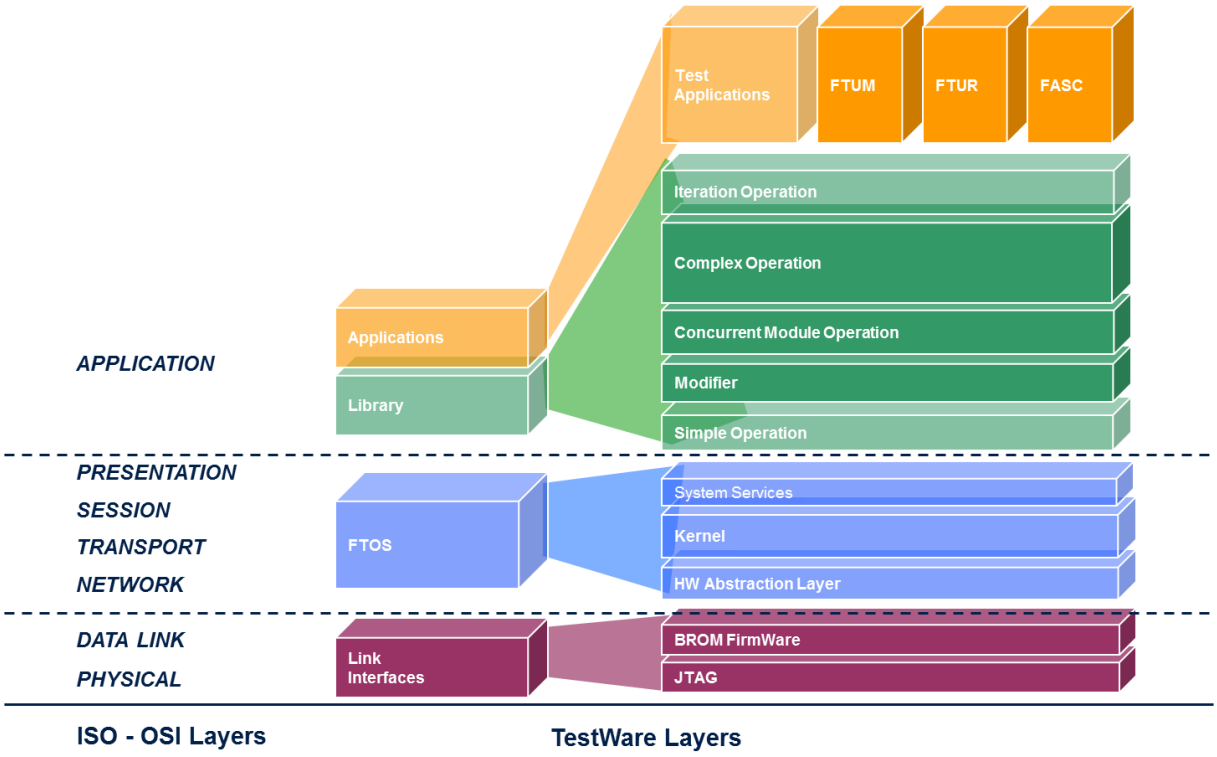


Figure 1: The latest version of testware layers

But first, what is functional testing? It is a quality assurance (QA) process and a type of black-box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is not theoretically considered (contrary to white-box testing). Now that

we know a definition of functional testing we can define what FTOS is: it is a software layer to interface ATE and DUT, which is responsible for the management of activities and resources. FTOS is loaded in RAM at the beginning of the test flow and it supports execution of product family oriented Test Applications (TAs) containing functions and algorithms used for both e-Flash production test and e-Flash analysis purposes. TAs releases are scheduled quite often due to constant test coverage improvement or test costs reductions. This is a trade-off in software testing because optimal test coverage need time for design these tests and execute them (time is money) but an optimal coverage guarantee low post-production cost avoiding defects. FTOS releases are less frequent: they have to follow only ATE or DUT new hardware requirements. We can hence say that FTOS was born to standardize execution of Infineon microcontroller e-Flash tests and is meant to support several Test Applications execution over different products belonging to the same family.
New generation products are introducing several cores into each microcontroller and memory space is rapidly increasing making the test time increase consequentially. These expansions are due to the increase of the power calculation demanded by the increase of the artificial intelligence into the cars and the space needed for processed data storing. In this sense the development of a multi-core version of the FTOS is required in order to:

- Reduce test time (thus reduce costs and upsurge the revenues) by making advantage of parallel memory access from different cores (timing requirement)
- Get closer to the actual multi-core usage of the costumer itself (test coverage requirement)

These two requirements justify the proposal of a multi-core OS.

FTOS is also a multi-task operating system. Multitasking permits to execute different processes simultaneously, alternating the execution of these. The change of process executed is called context switch, the switching decisions are taken by the scheduler and the dispatcher execute the switch operation. Multi-task permits to boost the throughput of the testing system.

1.3 Company's Methodology of Development

Infineon development process [2] is divided in 10 milestones which identify the current status of the product development. A milestone is the end of a stage that marks the completion of a work package or sub-phase, typically marked by a high level event such as completion, endorsement or signing of a deliverable document or high level review meetings. Five of these milestones are Business Gates (BG) that are management reviews of the business case and the technical risk assessment. They must have clear and visible criteria so that senior managers can make go/kill and prioritization decisions objectively.

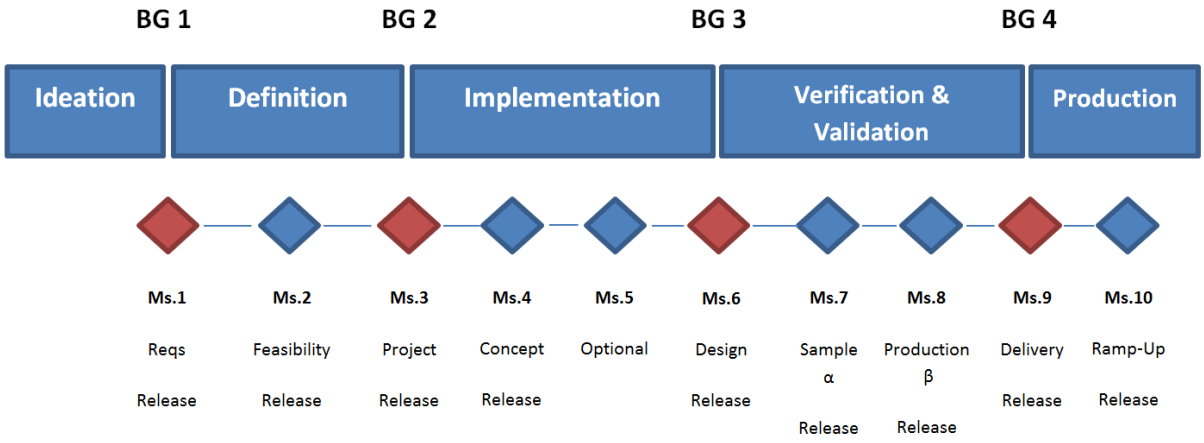


Figure 2: IFX Development Process

Across these milestones we can identify five main phases:

1. **IDEATION:** The costumer and the developers study a new idea usually based on client's needs, market predictions and competitor's benchmarks

2. **DEFINITION:** The idea is synthetized to a concept, which is a collection of all system macro requirements (REQs, in short). The concept requires several team reviews, redefinitions, proposal and finally ends up with a specification for the implementation. The specification is than proposed again to the costumer, which have to agree before the implementation.

3. **IMPLEMENTATION:** The specifications are then implemented by the developers, who realize a prototype of the desired product. Implementation constraints that crops up because were not considered in the specs definition may stop the process development and require a redefinition of the original concept.

4. **VERIFICATION & VALIDATION:** to validate a product means to check whether it works properly respecting all requirements and specifications, in all operating situations and with all possible boundary conditions (e.g. inputs, software loading and executing, high/low temperature, supply voltage range). If some requirements are not achieved, the product is pushed back to Implementation phase.

5. **PRODUCTION:** when validation BG is passed, the production stars, delivering the product to the final costumer

The passage across different BGs is not unidirectional. The product development state can cycle many times among all phases, requiring a great communication from one team to another.

1.4 Testware Development Process

Testware process development is based on V-Model for software development structure. V-Model [Figure 4] is an extension of the waterfall model [Figure 3]. This last is simpler, should be used for developing small projects, has no overlap phases and finally is not suitable for projects where REQs have high risk of changes. If there is a change in one intermediate step, the subsequent steps must be repeated until the end of the process.

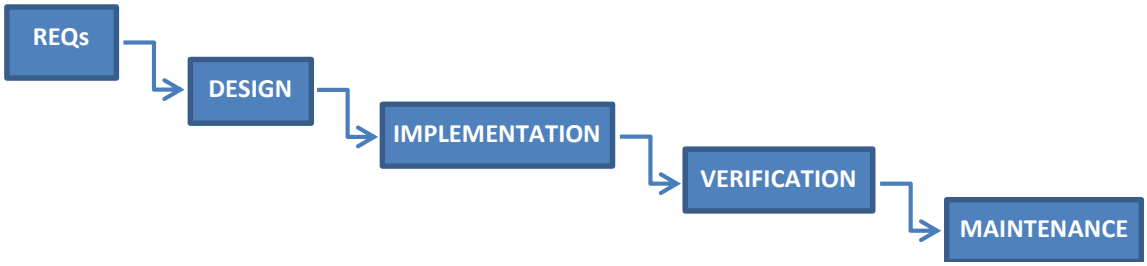


Figure 3: Waterfall model

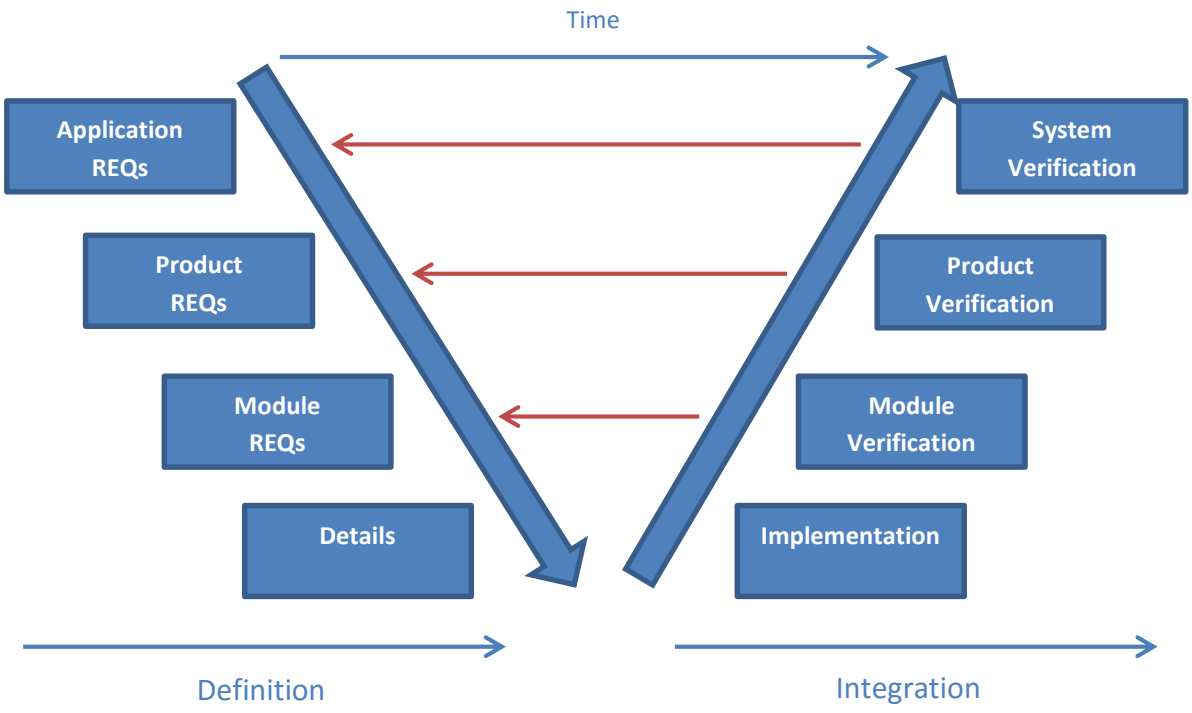


Figure 4: Testware V-Model

Instead of moving down in a linear way, the V-Model steps are bent upwards after the coding phase, to form the typical V shape. It is divided into:

1. **APPLICATION REQUIREMENTS**: defines the external requirements of the product to be implemented

2. **PRODUCT REQUIREMENTS**: defines the system behavior

3. **MODULE REQUIREMENTS**: defines the macro architectural blocks which compose the system

4. **DETAILS**: the single module behavior is described in details (aka LLD standing for Low Level Design)

5. **IMPLEMENTATION**: all modules are transcript into the specific programming language

Any phase has an associated phase of testing in the shape. This permits to have a proactive defect tracking and avoids the downward flow of defects. Software is developed during the implementation phase (no early prototypes of the software are produced) and gives equal weight to development and testing. This structure guarantee a stable and ordered way of work compared to agile approaches, losing in flexibility. For more details about this phrase see [3].

1.5 Tools for Operating System Release and Testing

The FTOS code is written in C programming language to guarantee a good level of abstraction and use also opcode functions to speed up the code execution. The OS is always committed to have a backup of what is done with the possibility of comparing actual version of code with the one committed. The testing apparatus is formed by a personal computer, an interface device and the testing board with the 32 bit Infineon TriCore™ microcontroller (the *Device Under Test*). With its real-time performance, embedded safety and security features, the TriCore™ family is a platform for a wide range of automotive applications such as the control of combustion engines, electrical and hybrid vehicles, transmission control units, chassis domains, braking systems, electric power steering systems, airbags and advanced driver assistance systems. TriCore™-based products also deliver the versatility required for the industrial sector, excelling in optimized motor control applications and signal processing. The pc with the interface devices forms the *Automatic Test Equipment*.

1.5.1 TASKING

It is the Altium microcontroller compiler for advanced automotive applications and it is used for writing code and compiling it. It supports different family of products, also the Infineon TriCore. It supports *C/C++* coding, provides a project files trunk, a console for log/error

messaging and, last but not least, the workspace where write the code. For more info see [4].

1.5.2 UDE

The abbreviation stands for Universal Debugging Environment and takes part in the interface devices of ATE. Debug sessions are part of the daily job during software development, due to the impossibility of never making mistakes. UDE software tool is the debugger used by the testware group before releasing new software. It supports most of Infineon products, using a standard JTAG connection and offers an easy access in read-write to RAM locations, windows to control directly CPU registers and memory locations, while new versions also allow a multi-core debug. Indeed cores can be set in halt/release state independently from the status of other ones, in order to better understand the execution of each CPU. Among other functionalities, a code profiler is provided, which can help in finding execution bottlenecks when unexpected slowdowns are detected. For more info see [5].

1.5.3 JAZZ

It is the Infineon microcontroller test harness. It is an instrument for testing and analysis, which can reproduce the test sequences of the ATE using a PC. JAZZ tool provides interface to device under test, typically with a JTAG access. Moreover, it can drive several control and measure instruments like power supplies, thermo-chambers or multimeters. JAZZ development is based on Object Oriented Programming concept which is a very powerful solution to get test reusability and portability. JAZZ tool can also import or export test patterns from and to other testers, following the direction of strict collaboration between product and test engineers. It is also based on a graphical user interface (GUI), easing user work and training, minimizing man-made errors probability. In JAZZ tool it is possible to create a dedicated tests collection for each device. Tests collection permits to build different test flows. Test flow is a tests collection subset to perform a certain analysis or characterization task; it can contain many copies of the same test and it must return always a "pass" or "fail" depending on result of each test.

2 Flash Memories

Since the OS presented is meant for e-Flash testing, an overview on the destination architecture and e-Flash model is needed in order to better understand the specific design choices and the case study.

2.1 Memories Introduction

For thousands of years, humans seek trickery to make less effort, which means evolve to a state of minimum energy. Manual tools, machinery, transportation equipment, industries, software tools, robots and so on are examples of trickeries that minimize the body energy consumption. A major reason of energy consumption of the human body is the brain: it represents only 2% of the weight of an adult but it uses 20% of the energy produced by the body [6]. Efficient energy supply is crucial for the mind so that our memory, mobility and senses can function normally. Then humans searched techniques for storing informations minimizing brain energy consumption and climb over the limits of manual storing. Today, almost every electronic device contains a memory device. Even in the automobile, electronic memories are becoming crucial, given the rise of artificial intelligence to be introduced in the control units of vehicles.

All memories, and in particular, Complementary Metal-Oxide-Semiconductor (CMOS) memories can be divided into two main categories: volatile and non-volatile. Volatile memories lose stored information as soon as the voltage supply is switched off; they require constant power to remain viable. Most types of Random Access Memories (RAM) fall into this category which can be further divided into Static-RAM (SRAM) and Dynamic-RAM (DRAM). On the other hand, memories that maintain their data when the power supply is removed are called Non-Volatile Memories (NVM). The first type of NVM was implemented by writing permanently the data in the memory array during manufacturing (mask-programmed Read Only Memories, ROM). A big step in CMOS memories was made when Erasable Programmable Read Only Memories (EPROM) were introduced, which can be electrically programmed and erased by exposing them to Ultra-Violet (UV) radiation for about 20 minutes. Electrically Erasable Programmable Read Only Memories (EEPROM) are electrically erasable and programmable, but require more area on silicon to be implemented, lowering density of the memory itself. Flash memories are non-volatile memories in which a set of cells can be electrically programmed and a large number of cells (block, sector or page) are electrically erasable at the same time. So erase operation is very fast since the whole memory can be erased in a single operation. One of the major applications for Flash memories is their integration inside SoC to allow software updates,

reconfigure the system, and allow non-volatile storage; Flash memories which are integrated in SoC are generally called embedded-Flash memories (e-Flash). The two fundamentals parameters of a non-volatile memory are:

1. **ENDURANCE**: the capability of keeping stored informations are a huge number of erase/program/read cycles
2. **DATA RETENTION**: the capability of keeping the stored informations after a big lapse of time

The focus of this big overview is the embedded flash memory, the type of memory tested at Padua Development Center of Infineon Technologies AG.

2.2 Single Flash Cell

To have a memory cell that has two logical states and maintains its stored information independently of external conditions, the storage element needs to be a device whose conductivity can be altered in a non-destructive way and when it is turned off must keep the charge inside. In CMOS Flash memories, this is achieved changing the threshold voltage V_T of the transistor, by making it lower or higher of a predefined value, and thus identifying the two logical states of a "programmed" or "erased" cell. This threshold voltage can be described as

$$\mathrm{V_T} = \mathrm{K} - \frac{\mathrm{Q}}{\mathrm{C_{ox}}}$$

where

- K is a constant that depends on gate and substrate material, channel doping and gate oxide capacitance
- Q is the charge in the gate oxide
- C_{ox} is the oxide capacitance between CG and FG

By modifying the charge trapped in the gate oxide, we can create a "shift" in the I-V curve of the transistor and then codify a right-shift with a logical state and a left-shift with the other logical state. Usually "0" states for "uncharged" (erased) and "1" as "charged" (or programmed) as it can see on [Figure 5].

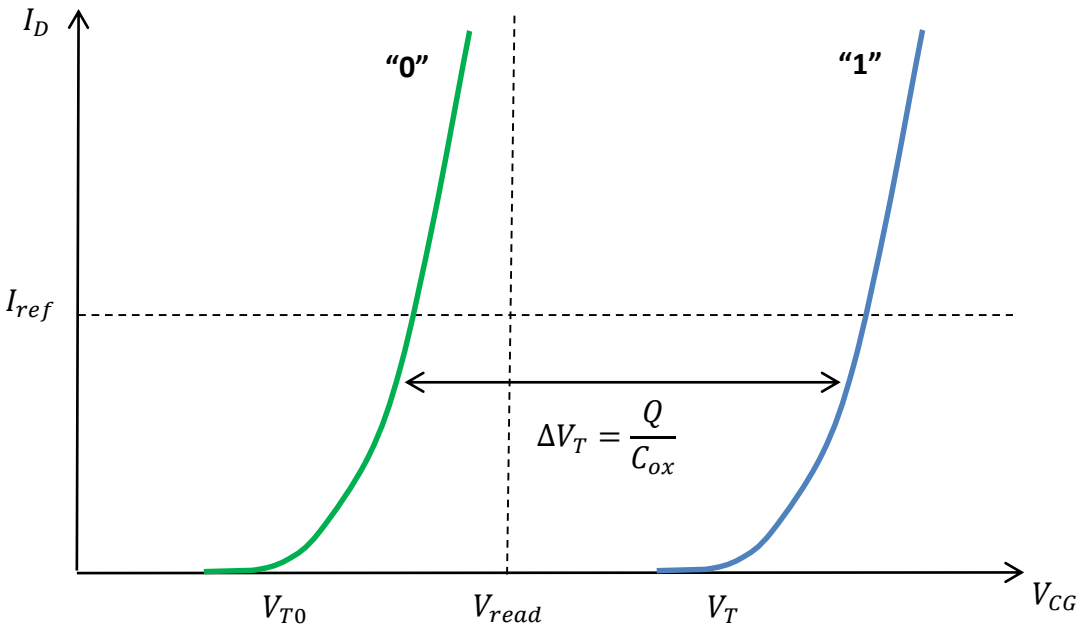


Figure 5: I-V curves shifting for uncharged/charged cases

The technology adopted in the e-Flash memory used in the SoC is the FGMOS where the charge is stored in a conductive layer that is between the gate and the channel and is completely surrounded by insulator.

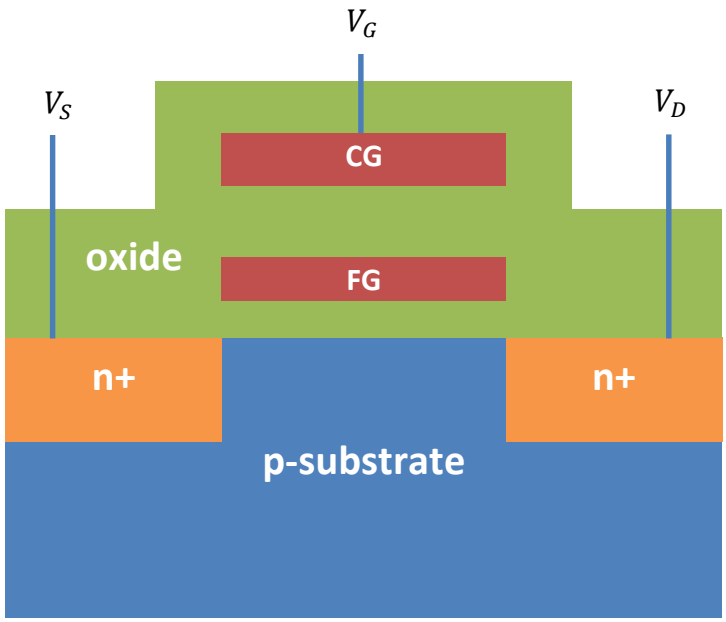


Figure 6: FGMOS cross section

The [Figure 6] shows the cross-sectional view of a flash cell. On top of a flash cell is the control gate (CG) and below is the floating gate (FG). The FG is insulated on both sides, on top by an inter-poly oxide layer and below by a tunnel oxide layer. As a result, the electrons

programmed on the floating gate will not discharge even when flash memory is powered off. The voltage applied on the CG generates and controls the conductivity of the conductive channel between the source and the drain electrodes. The minimum voltage that can turn on the channel is the threshold voltage. As we can see in the V_T equation, if are injected electrons between CG and FG the threshold voltage increase: this means that the cell is charged. Programming is done on one bit (or byte) at a time, while erasure is done on all cells in the same memory block. The saturation region for a conventional MOS is where I_{DS} is essentially independent of the drain voltage. In a FGMOS transistor, the drain current will continue to rise as the drain voltage increases and saturation will not occur. But, how it can be checked if a bit cell is programmed or erased? Simple, if charge is stored in FG, it is possible to measure the drain current I_D of the cell with a fixed V_{GS} voltage. If this current is over a reference current, the cell results erased ($Q_{FG} = 0$) and if it is under the reference value the cell can be considered as programmed. Storing/removing FG charge is commonly done using one of these mechanisms:

- Channel Hot Electron Injection (CHEI)
- Fowler-Nordheim (FN) tunneling

The choice of program/erase mechanism depends on the bit cell structure, array organization, and process technology. The physical mechanism of CHEI is relatively simple to understand qualitatively. An electron traveling from the source to the drain gains energy from the lateral electric field and loses energy to the lattice vibrations (acoustic and optical phonons). At low fields, this is a dynamic equilibrium condition, which holds until the field strength reaches approximately $100\ kV/cm$. For fields exceeding this value, electrons are no longer in equilibrium with the lattice, and their energy relative to the conduction band edge begins to increase. Electrons are "heated"[2] by the high lateral electric field and a small fraction of them have enough energy to surmount the barrier between oxide and silicon conduction band edges. For an electron to overcome this potential barrier, three conditions must hold:

1. Its kinetic energy has to be higher than the potential barrier
2. It must be directed toward the barrier
3. The field in the oxide should be collecting it

If these conditions are satisfied, the electrons are injected by the high electric field in the FG. This is a power consuming mechanism due to the large currents and low injections efficiency. The FN tunneling instead is based on quantum mechanics. The solutions of the

[2] Refers to the effective temperature term used when modelling carrier density (i.e., with a Fermi-Dirac function) and does not refer to the bulk temperature of the semiconductor (which can be physically cold, although the warmer it is, the higher the population of hot electrons it will contain all else being equal). The term "hot electron" was originally introduced to describe non-equilibrium electrons (or holes) in semiconductors

Schrodinger equation represent a particle. The continuous nonzero nature of these solutions, even in classically forbidden regions of negative energy, implies an ability to penetrate these forbidden regions and a probability of tunneling from one classically allowed region to another. This occurs with the presence of a high electric field. The FN tunneling method is widely used in NVM. The reasons for this choice:

- Tunneling is a pure electrical mechanism
- The involved current level is quite low
- It allows to obtain a program time shorter than retention time

But, on the other hand, the exponential dependence of FN tunnel current on the oxide field causes critical problems. A small variation of oxide thickness t_{ox} among the cells in a memory array results in a great difference in programming or erasing currents thus spreading the V_T distribution. Therefore, oxide defects must be avoided to control program/erase characteristics and obtain a good reliability.

A common problem for CHEI and FN tunneling is the high electric filed used for programming the cells. The negative charge trapped into the floating gate decrease the electric field across the gate oxide, so that at the increasing of cell living time it is more and more difficult to inject charge in the FG. Besides, for sustain high electric fields is requested a good doping profile shaping. For more detailed info about single flash cells see [7] , [8] and [9].

2.3 Flash Memory Architecture

Two main technologies dominate the non-volatile flash memory market today: NOR and NAND. Both NOR and NAND flash memories were invented by Dr. Fujio Masuoka while working for Toshiba around 1980. The name "flash" was suggested by Dr. Masuoka's colleague, Mr. Sho-ji Ariizumi, because the erasure process of the memory contents reminded him of the flash of a camera [10]. NOR flash was first introduced by Intel in 1988. There are two main types of flash memory, which are named after the connection in a way that resembles NAND and NOR logic gates, as NAND and NOR flash memories. The NOR Flash memory is the most commonly used in a wide range of applications that require both medium density and performance. This is the memory architecture of our SoC. In a NOR matrix organization cells are arranged in rows (called word lines) and columns (called bit lines): all the gates of the cells in a row are connected to the same word line (WL), while all the drains of the cells in a column are connected to the same bit line (BL); the source of all the cells in the sector are connected to a common source line (SL). The read operation is done by byte or by word; therefore one cell for each output is addressed. There are some methods to identify the cell status but the most commonly used is to compare the current of the cell with the one of a reference current; the result of the comparison in then converted

into a voltage which is fed to the output. The Sense Amplifier converts this small voltage level to the external higher level. Like read, also program operation is generally performed by byte or by word. The array of cells is physically divided in different sector, each one erasable separately by means of a dedicated source switch. Sectors can be equal or of different size.

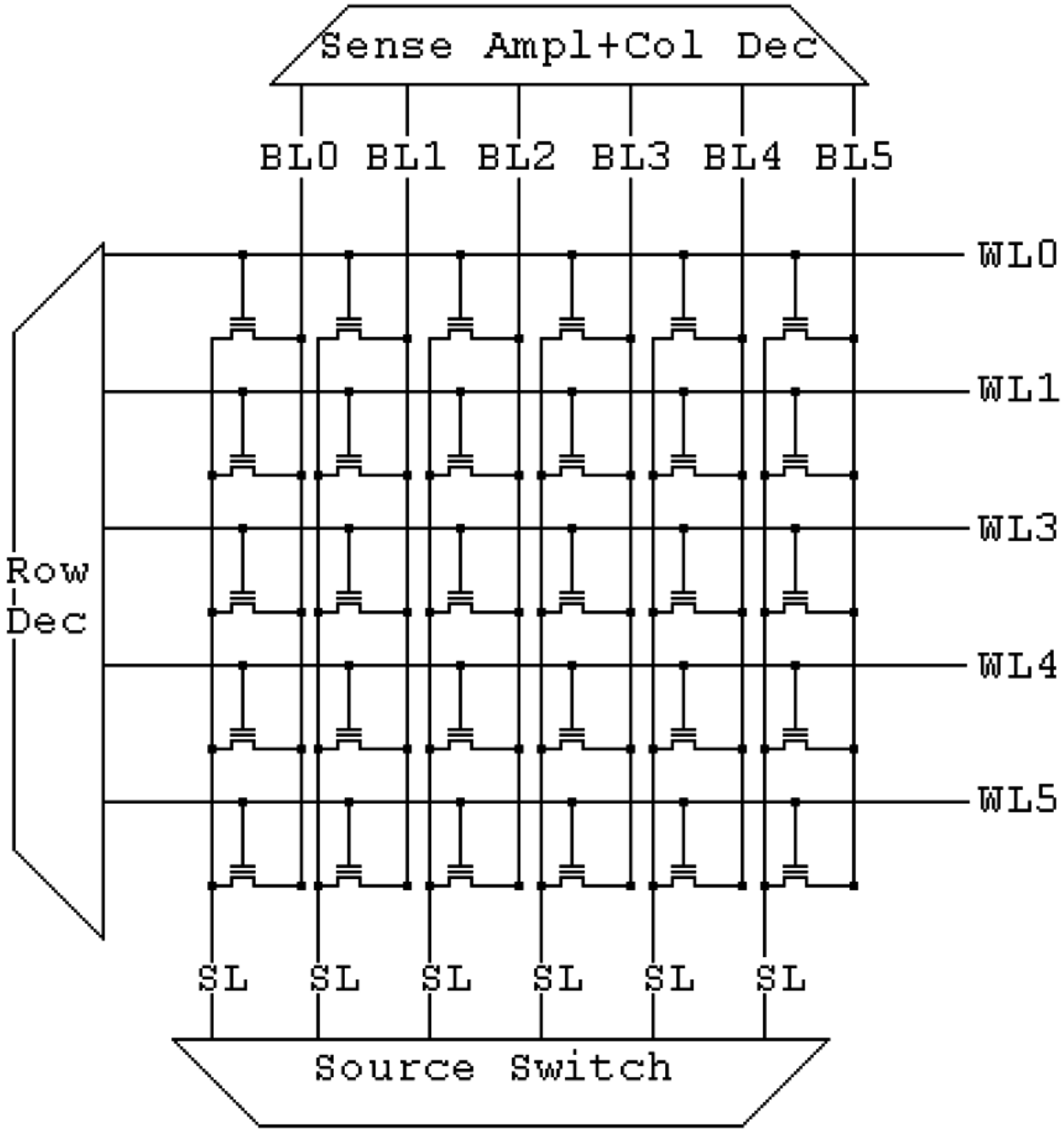


Figure 7: NOR flash matrix with decode circuitry and sense amplifiers

It is helpful to divide the memory into blocks, banks and sectors to reduce the delays over the wordlines and bitlines traces. The decoder has the task to enable rows and columns that must be accessed by the externa circuits.

The NAND Flash memory is similar to the NOR, but the access to the matrix is different because the cells are arranged into the array in serial chains: the drain of a cell is connected with the source of the following one. The elementary unit of a NAND architecture flash memory is not a single cell but a serial chains of more FG transistor connected to the bit line and ground through two selection transistors. This organization permits to eliminate all contacts between Word Lines (WL), reducing in this way the occupied area. The reduction of the matrix area is the main advantage of this solution. Selection transistors are biased to connect the chain to the bit line and isolate it from the ground. If the memory is organized in a NAND array, both program and erase mechanism are electrons tunneling. Since tunneling is more power efficient than CHE injection, currents are smaller and different supply voltages can be internally generated by charge pumping circuits implemented in the same die. NAND array are preferred for high-density Flash memories. During reading operation the selected cell has the control gate at 0V while the other cells in series are driven at high voltage, thus acting as ON pass gates independently of their actual thresholds. The current, which flows through the series only if the selected transistor presents a negative threshold, can be detected by the sense amplifier, which can interpret the stored data.

2.4 Flash Memory Defects and Tests

A defect is a physical anomaly in the circuit's material, which can occur due to many factors in every manufacturing process. Defects can be characterized as a short, an open, increased resistance or capacitance and so on. A fault is the logical representation of a defect. Not all defects lead to faults. Some flash memories defects are listed below.

For the single cell:

- t_{ox} oxide thickness variation
- WL to FG resistance
- FG to drain resistance
- FG to source resistance
- CG to drain resistance
- Source and drain to bulk resistances
- Source to drain resistance
- CG, drain, bulk opened
- Mobility reduced

For the word line:

- WL opened
- WL to bulk resistance

For the sectors:

- WL i-th to WL (i+1)-th resistance
- Local BL opened
- Local SL opened
- Local BL i-th to local BL (i+1)-th resistance
- Local BL i-th to local SL (i+1)-th resistance

For the memory blocks:

- Global BL opened
- Global SL opened
- Global BL i-th to global BL (i+1)-th resistance
- Global SL i-th to global SL (i+1)-th resistance

For the decoders:

- Same WL of two sectors selected
- Multiplexer incorrect output

For the sense amplifiers:

- Wrong reference current, i.e. not good sensing operation

These defects occurs after the manufacturing and must be tested how many are and correct it if it is possible. For this purpose, redundancy banks are often used in e-Flash memories: those banks are not used unless a defect is detected in the memory array. In this case, the failing BL/sector is logically replaced with a redundancy one, saving the SoC from being discarded. The mapping of redundancy bitlines is saved into a particular area location, and it is configured during the automated test flows.

The test and characterization engineers must take into account the list of some aspects reported below:

- PE[3] performance
- PE disturbs
- Retention
- Endurance
- DC test/parametric

When a cell is programmed (or erased) the threshold voltages are different depending on silicon process variations. The distribution of the threshold voltages for erased and programmed cells can compromise the operation of storing and reading information inside the memory array. As it can read on page 2 in [11], the V_T voltage of erased memory cells tends to have a wide Gaussian-like distribution:

$$p_E(x) = \frac{1}{\sigma_E\sqrt{2\pi}}\, e^{-\frac{(x-\mu_E)^2}{2\sigma_E^2}}$$

Where μ_E and σ_E are the mean and standard deviation of the erased state threshold voltage. A test which verifies the correct writing of 0s and 1s patterns can detect anomalous distribution of threshold voltages. Flash memory PE cycling causes damage to the tunnel oxide of floating gate transistors in the form of charge trapping in the oxide and interface states which directly results in threshold voltage shift and fluctuation and hence gradually degrades memory device noise margin. Major distortion sources include:

- Electrons capture and emission events at charge trap sites near the interface developed over PE cycling directly result in memory cell threshold voltage fluctuation, which is referred to as random telegraph noise
- Interface trap recovery and electron detrapping gradually reduce memory cell threshold voltage, leading to the data retention limitation

Moreover, electrons trapped in the oxide over PE cycling make it difficult to erase the memory cells. On [Figure 8] we can see the Gaussian distributions of threshold voltages. The sketched lines represent the drain-source currents: when the cell is erased the FG is uncharged and all the charge is in the channel; for this reason the current is higher than I_H. When the cell is programmed the FG is charged and only a part of the total charge is in the channel; for this reason the current is lower than I_L. The Gaussian distributions (bold lines) are centered in the respective mean threshold voltages μ_E.

[3] **PE** stands for **Program/Erase**. Flash memory cells gradually wear out with the PE cycling which is reflected as gradually diminishing memory cell storage noise margin (or increasing raw storage bit error rate).

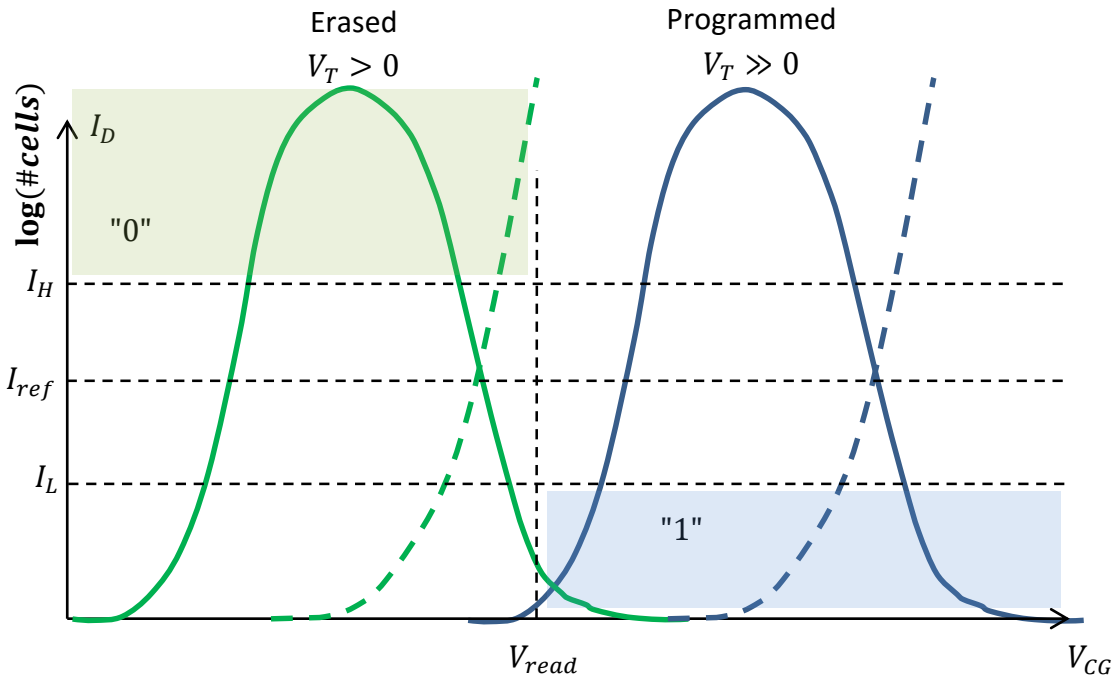


Figure 8: Statistical distributions of V_T during erase and program operations

A test which verifies the correct writing of 0s and 1s patterns can detect anomalous distribution of threshold voltages. Reference currents for the read operations define the tolerance in the variation of threshold voltages, and are generally set differently for erase and program operations, as shown in figure. A sweep on the read currents is performed for the characterization of the flash memory and can provide useful information about “how well” or “how bad” cells are programmed or erased (margin sweep test). Likewise a test which modifies the Control Gate voltages for the read operations can provide similar information (CG scan).

The failure mechanisms referred to as program disturb concern data corruption of written cells caused by the electrical stress applied to these cells while programming other cells in the memory array. Considering a NOR array of flash cells, if we want to program a cell, a high voltage applied to the WL and a negative voltage to the BL are necessary to permit FN tunneling. In these bias conditions there are two major disturbs. The first one, due to the high positive voltage applied to the WL, is called Gate Disturb. This kind of disturbs stress FGMOS which have their gate connected to the WL. There might be tunneling through the oxide and so it is possible to put some charge into the FG of transistors that are not selected. In the second kind of disturb, called Drain Disturb, a relatively high voltage applied to the selected BL can stress the drain of all the cells whose are in the same bit line of the cell to program. In this way, a loss of the charge trapped into the FG could happen. The test flow of a Flash memory, have also to detect if Gate Disturbs and Drain Disturbs can compromise the information stored into disturbed bitcells. Programming *Checkerboard* [Figure 9] or *Zebra patterns* can help the tester to detect these stress conditions. A *checkerboard pattern* [12]

alternates 0s and 1s in both rows and columns of the array, while a *zebra pattern* alternates columns of all 1s with columns of all 0s. The pattern must be programmed at physical level and not at logical level.

Retention errors are value dependent; their frequency is asymmetric with respect to the value stored in the flash cell. Examples of retention errors are 00→01, 01→10, 01→11 and so on. During retention test, the electrons stored on the floating gate gradually leak away under stress induced leakage current. When the floating gate loses electrons, its $V_T$shifts left from the state with more electrons to the state with fewer programmed electrons. For more details [13].

As blocks are repeatedly erased and programmed the oxide layer isolating the gate degrades. This reduces the endurance of the e-Flash memory. Program/erase endurance can be tested by repeatedly programming a single page with all 0s (vs. the erased state of all 1 bits), and then erasing the containing block; this cycle can be repeated until a program or erase operation terminated with an error status [14]. This parameter is measured as number of PE cycles.

DC tests/parametric are contacts tests (opens and short checks), power consumption tests, leakage tests, threshold tests (max and min input voltage at which the device switch from high to low), current tests, timing measurements (rise and fall time, delay, access time measurements), Schmoo plots, etc.

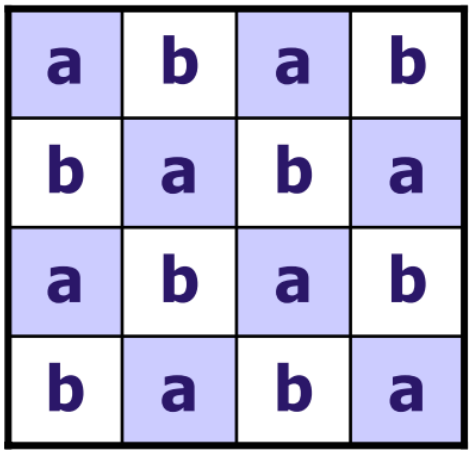


Figure 9: Example of checkerboard pattern where *a* and *b* are 0 and 1

3 System on Chip Architecture

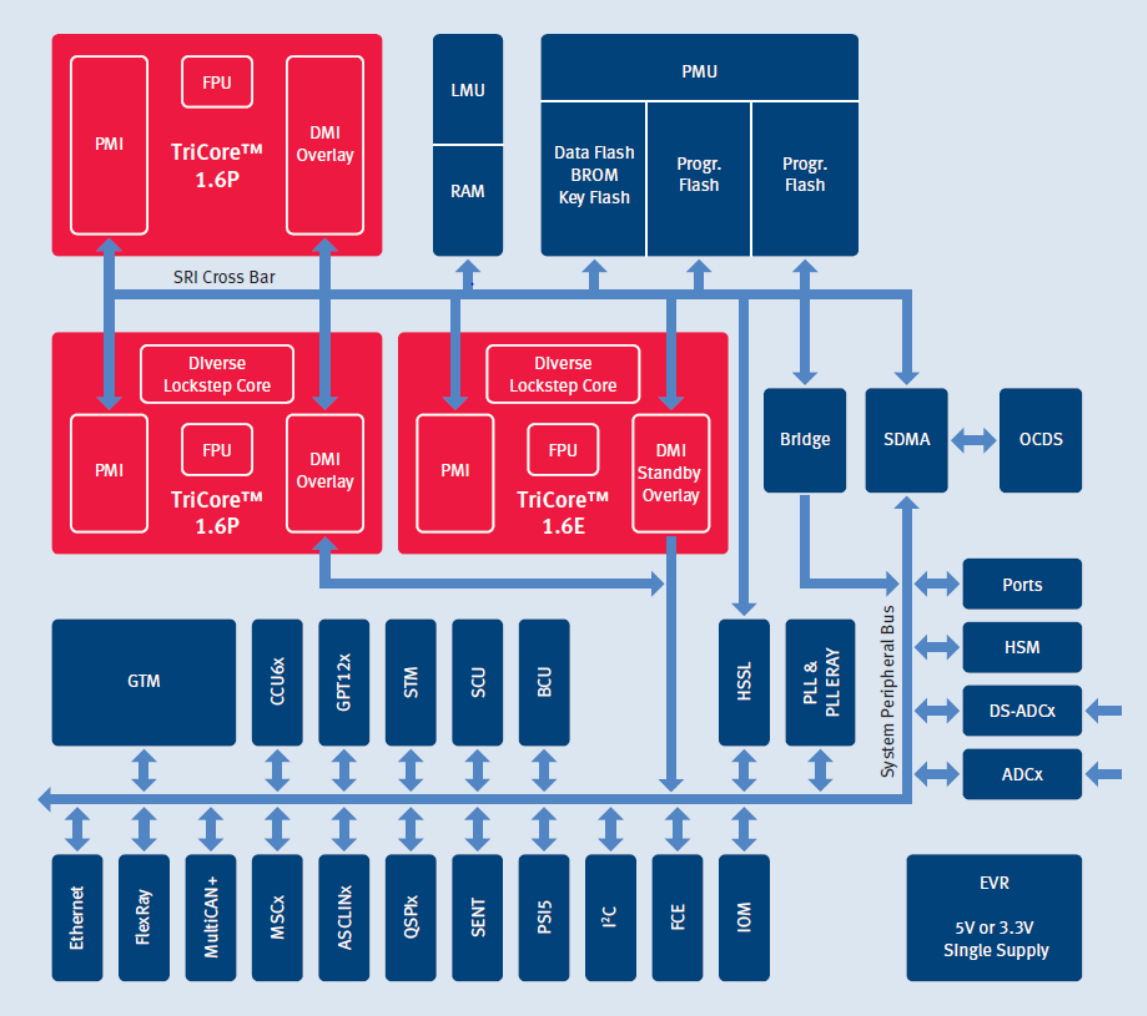


Figure 10: Aurix™ system architecture

Aurix is Infineon's family of microcontrollers serving the needs of the automotive industry in terms of performance and safety. Some applications, where AURIX™ is used, are: chassis domain control, ADAS[4], gasoline engine management and the transmission control unit (TCU). AURIX™ permits to achieve the ASIL-D level. ASIL stands for *Automotive Safety Integrity Level* and refers to the highest classification of initial hazard (injury risk) defined within ISO 26262 and to that standard's most stringent level of safety measures to apply for avoiding an unreasonable residual risk [15].

[4] Advanced Driver Assistant Systems is the group of safety mechanisms like lane assist, emergency brake assist, distance control, etc. Radar technology collects the information in and around the vehicle and the microcontroller elaborate the informations.

It has a multi-core architecture that is based on three 32-bit TriCore™ CPUs running at 200 MHz in the full automotive temperature range. The block diagram of the presented SoC is showed in [Figure 10]. The 3 red blocks are the CPUs in the SoC. The TC27x sub-family includes two high performance TriCore TC1.6P CPU cores and one high efficiency TriCore TC1.6E CPU core. The two cores are slave cores and the high efficiency core is the master core [16].

In this chapter will be reported, synthetically, some selected features of the SoC needed to understand the operating system release and the validation.

3.1 Internal Buses

The SoC has two independent on-Chip buses:

- **SRI** crossbar[5], that is the Shared Resources Interconnect protocol that connects the TriCore CPUs, the high bandwidth peripherals and the Direct Memory Access module (DMA) to its local resources for instruction fetches and data accesses
- **SPB** (System Peripheral Bus) that connects the TriCore™ CPUs, the high bandwidth peripherals and the DMA module to the medium and low bandwidth peripherals

For our intents we use the SRI crossbar (X-bar) that supports parallel transaction between different SRI-Master and SRI-Slave peripherals, which are both referred as Agents of the SRI bus. The SRI X-bar supports also pipelined requests from the SRI-Master interfaces. This is the first important capability of the presented SoC, which actually allows us to consider the implementation of a Multi-Core OS in which a parallel access to the e-Flash memory is desired. The arbitration of the SRI X-bar supports atomic[6] transfers that are generated by atomic assembly instructions which require two single transfer instructions of read and write.

Due to the support for parallel transactions, an arbiter module, on [Figure 11], is implemented for each SRI-Slave, as shown in the figure. The operating frequency is called f_{SRI} and is generated separately from the Clock Control Unit (CCU).

[5] In analogy with old electromechanical telephony crossbar switches
[6] Uninterruptable read-modify-write memory operations limited to specific functions and data size

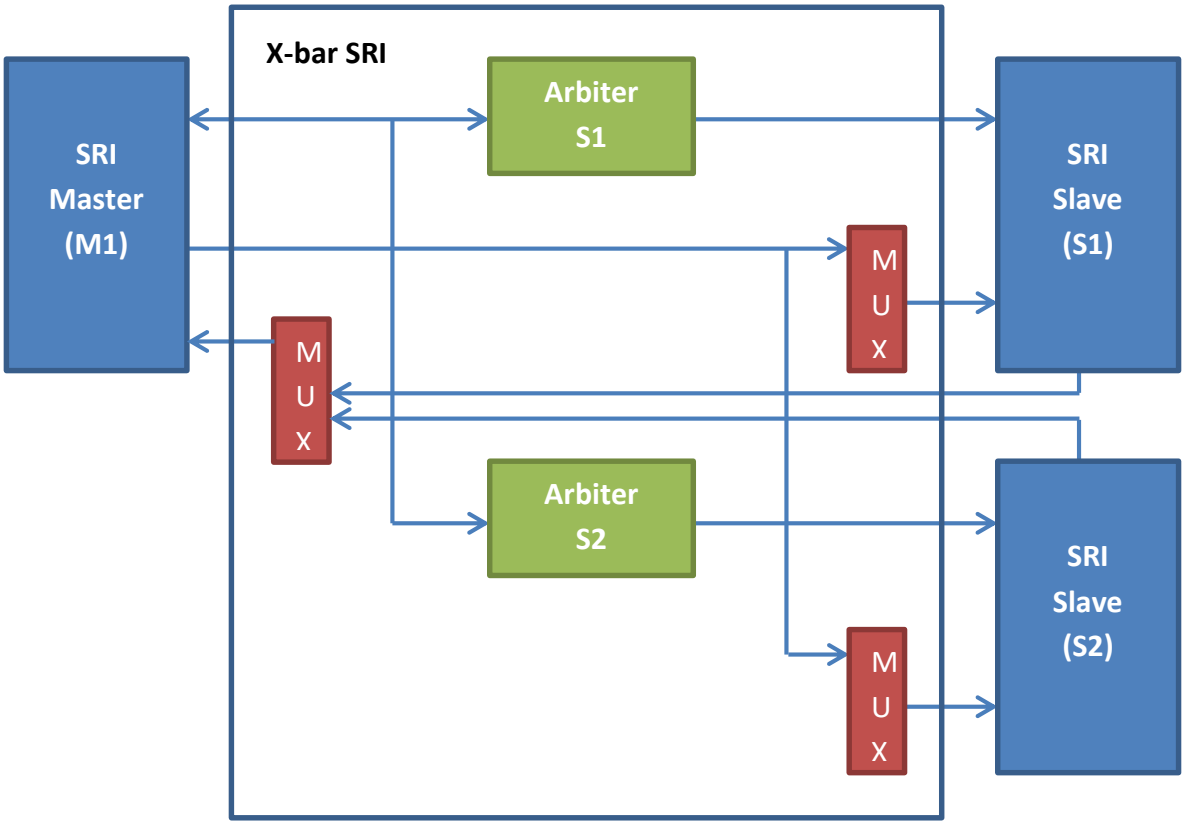


Figure 11: SRI arbitration module

3.2 On-Chip Debug Controller

The SoC provides the infrastructure for the various tools used during the development and maintenance of the application. A fundamental tool is the internal debugger: in this situation the application is not yet ready, the system outside of the microcontroller is either not connected or under control by other means so that misbehavior of the software has no catastrophic consequences. In this condition, the user is a software engineer with thorough knowledge of the device and the system, in other words, no protection is needed. From the debug tool the user expects:

- **Download capability**: The memories of the SoC (and of other external memories attached to the SoC) must be written (and programmed in case of non-volatile memories) without need to disassemble the application system

- **Running Control**: Each processor core can be stopped and started at will, either separately or synchronously throughout the SoC

- **Visibility/Writability**: The content of all storage locations inside the SoC, i.e. memories, SFRs and processor registers, can be read and written, preferably even while the system is running

- **Traceability**: A log of the processing is desired, as detailed ("cycle accurate") and wide (aligned trace of parallel processes) as possible

These capabilities are mainly managed via the *On-Chip Debug Support* infrastructure (OCDS). Of particular interest are the Run Control features, by which the running CPUs are dynamically configured via access of dedicated Debug Status Register (DBSR). The internal implementation of the SoC sets only the CPU0 in run mode at startup, and puts other CPUs in a Halt-After-Reset state. Halted CPUs have to be released by user (via Debug control features, see [Paragraph 1.5.2]) or by the running CPU with specific debug control instructions. Of course a state-aware watchdog timer suspension during debug sessions is implemented.

3.3 MMU

As the FTOS is oriented to test the Flash Memory Unit of the presented SoC, a deeper insight of the Memory Management Unit (MMU) is required. Furthermore, since the FTOS code is downloaded into the RAM memory, we will also have a look at both volatile and non-volatile memories contained in the SoC.

The TriCore microcontroller has the following CPU related memories:

Program Memory Unit (PMU0) with:

- 4 MB of Program Flash Memory
- 384 KB of Data Flash Memory
- User Configuration Blocks (UCB)
- 32 KB of Boot ROM (BROM)

CPU0 with:

- 24 KB of Program Scratch-Pad SRAM (PSPR)
- 112 KB of Data Scratch-Pad SRAM (DSPR)
- 8 KB of Program Cache (PCache)

CPU1 with:

- 32 KB of Program Scratch-Pad SRAM (PSPR)
- 120 KB of Data Scratch-Pad SRAM (DSPR)
- 16 KB of Program Cache (PCache)
- 8 KB of Data Cache (DCACHE)

CPU2 with:

- 32 KB of Program Scratch-Pad SRAM (PSPR)
- 120 KB of Data Scratch-Pad SRAM (DSPR)
- 16 KB of Program Cache (PCache)
- 8 KB of Data Cache (DCACHE)

The system has non-uniform memory access timing (NUMA), i.e.:

Fast (1 cycle typical)

- Read/Write to local Data Scratch (DSPR) or DCache
- Fetch from local Program Scratch (PSPR) or PCache

Medium (6-8 cycles)

- other DSPRs/PSPRs (including data in local PSPR and fetch from local DSPR)

Slow (8-20 cycles)

- Program Flash

Very Slow (> 20 cycles)

- Data Flash (EEPROM + UCB)

Boot ROM is the memory sector that permits the startup of the system. Scratchpad RAMs are high-speed internal memories used for temporary storage of calculations, data, and other work in progress. They are used to hold small items of data for rapid retrieval and are mostly suited for storing temporary results (as it would be found in the CPU stack) that typically wouldn't need to always be committing to the main memory. In particular PSPR (Program Scratchpad RAM) stores codes and DSPR (Data Scratchpad RAM) stores data. Each

RAM segment can be accessed in an absolute addressing mode, or with a relative addressed mode. Using the relative address mode, each CPU can access only its own PSPR and DSPR, and this feature is useful to make some particular data/code private for each CPU (i.e. not accessible from other cores). As a consequence, a CPU trying to access a private data/code, different from its own one, will trap into an illegal instruction or load a wrong content. In order to increase performances, every data/code frequently used by a CPU should be located inside the relative RAM (PSPR or DSPR), reducing the penalty for accessing far memory location. The *cache* instead, is a component that stores data so future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation, or the duplicate of data stored elsewhere. A cache hit occurs when the requested data can be found in a cache, while a cache miss occurs when it cannot. The caches are usually very small to be cost-effective and to enable efficient use of data. Naturally PCache stands for Program Cache and DCache for data cache.

Also the flash module is divided into program and data sections:

- Program flash 0
- Program flash 1
- Data flash 0
- Data flash 1

Each bank is further divided in Physical Sectors which contain a different number of Logical Sectors. Logical Sectors are divided in word-line clusters, word-lines and then in pages. The benefit of having sectors is that the Flash memory is sector-erasable, meaning you can erase one sector at a time. In the past, erase commands erased the entire memory chip - therefore to keep a working copy of that data during run-time, an application required additional memory [17]. For reliability reasons, each page has a subset of Error Correction bits, which guarantee the detection and correction of failing bits, referred as ECC. This expensive correction mechanism is fundamental in automotive applications, where errors must be avoided for safety reasons. Flash memory consists also of some redundancy sections, as already mentioned. Whenever a bit-line has a process defect, it is logically replaced (i.e. mirrored) with a new bit-line of the redundancy section. The PMU is connected with the SRI X-bar via dedicated SRI Ports on the Program Flash 0, Program Flash 1, and Data Flash [Figure 12]. That means that a concurrent access is possible for PF0 and PF1, but not for DF0 and DF1 (it should need between arbitration master agents, introducing some access delay). The operating frequency of the PMU is generated by the CCU, which also provides the clock for all modules inside the SoC. Low level instructions on the flash triggered by the CPU (such as read, program, erase, ECC check, etc.) are executed by the internal logic of the PMU and by a finite state machine.

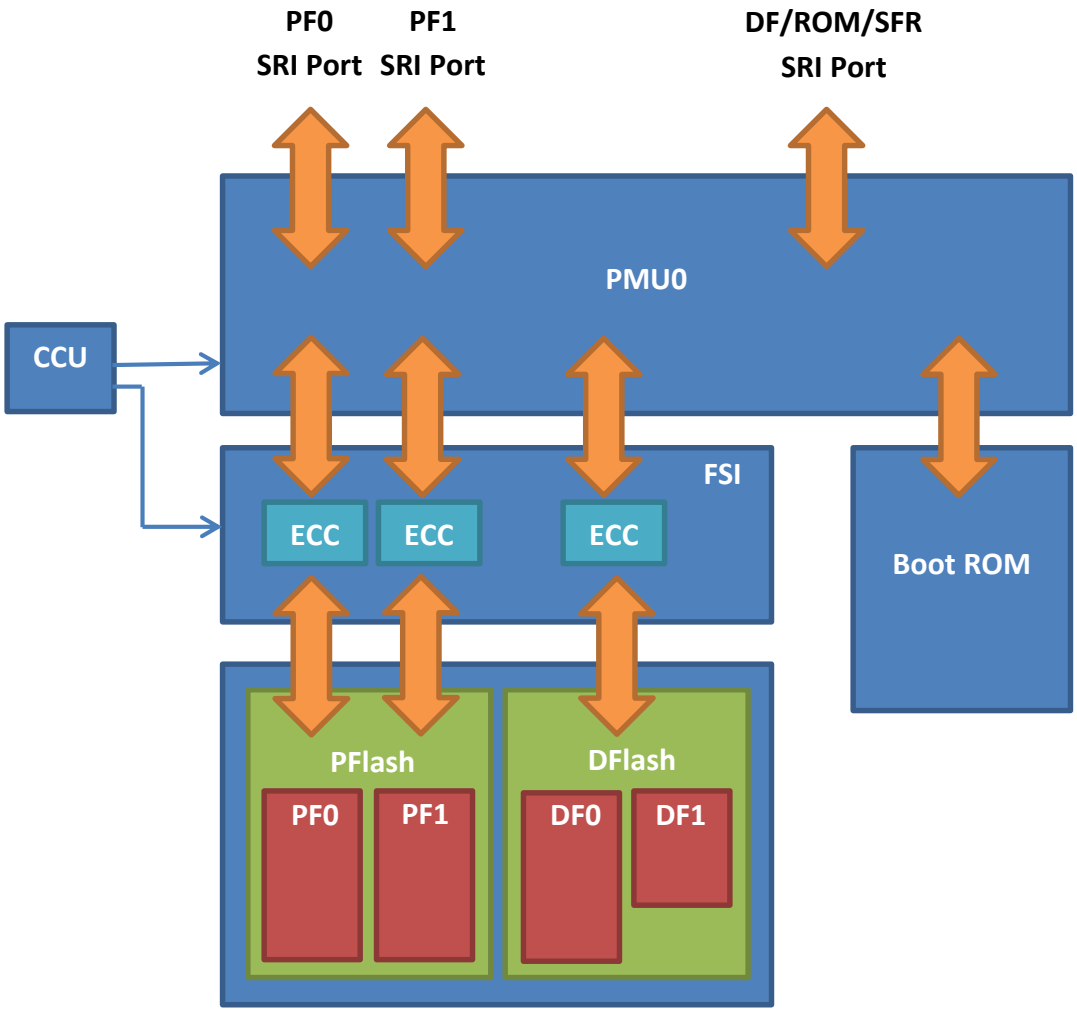


Figure 12: PMU synthetic scheme

3.4 TriCore™ Core Architecture Overview

TriCore™ is a 32-bit DSP and microcontroller with single-core architecture optimized for real-time embedded systems. The TriCore™ Instruction Set Architecture (ISA) combines the real-time capability of a microcontroller, the computation power of a DSP and the high performance/price ratio of RISC architectures, in a compact re-programmable core. The ISA supports a uniform, 32-bit address space, with virtual addressing capabilities and memory-mapped I/O. The architecture allows for a wide range of implementations, ranging from scalar through to superscalar, and is capable of interacting with different system architectures, including multiprocessing. The architecture supports both 32-bit and 16-bit instructions formats (as a subset of the 32-bit instructions chosen by their frequency of use) to reducing code space occupation, lowering consequently memory requirements and power consumption. Typical DSP instructions and data structures are largely supported by seven addressing modes used, for instance, in Finite Impulse Response (FIR) filters, or in FFT calculation. They also support efficient compilation of C/C++ programs.

Real-time responsiveness is largely determinate by interrupt latency and context-switch time[7]; the high-performance architecture minimizes the interrupt latency by avoiding multi-cycle instructions and by providing a flexible hardware-supported interrupt scheme which also grants a fast-context switch.

The key features are here summarized:

- 32-bit architecture
- 4GBytes of address space (physical and virtual)
- 16/32-bit instructions for reduced code size
- Most instructions executed in one cycle
- Branch instructions with branch prediction module
- Low interrupt latency with fast context-switch using wide pathway to on-chip memory
- Zero-overhead loop capabilities
- Floating-Point Unit (FPU)
- Memory Management Unit (MMU)
- Single bit handling capabilities
- Flexible interrupt prioritization scheme
- *Little-endian* byte ordering for data memory and CPU registers
- Memory protection
- Coprocessor support
- Debug support

The main topics of interest will be the Context-Management System, supported by an Infineon international patent, the System Timer Module (STM), Interrupts and Traps management. The registers are divided into:

- 32 General Purpose Registers (GPRs)
- Program Counter (PC)
- Two 32 bit registers containing status flags, previous execution informations and protection informations: Previous Context Information (PCXI) and Program Status Word (PSW)

PCXI, PSW and PC are fundamental for storing and restoring task's context.

[7] In order to be classifiable as an RTOS an operating system must: have a time predictably response and be deterministic. For these reasons interrupt latency and context-switch time must be minimized to have the sureness of execute the processes in a determinate fixed times, imposed by the work context.

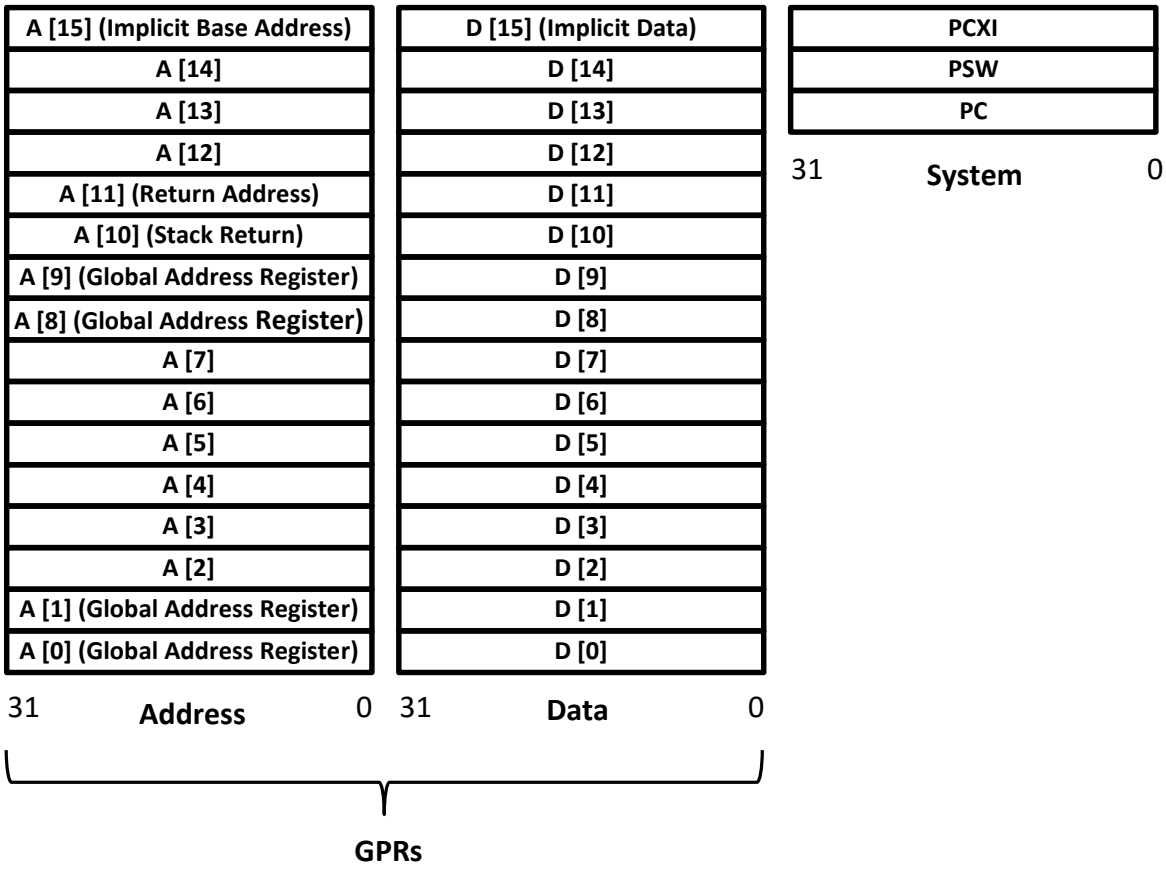


Figure 13: TriCore™ context registers

The 32 GPRs are divided into sixteen 32-bit data registers (D [0] to D [15]) and an equal number of address registers (A [0] to A [15]). Architectural registers, together with the Context Management registers, compose the *context of a task*. Registers [0_H - 7_H] are referred to as the Lower Context registers and registers [8_H - F_H] are called Upper Context registers. In addition to the GPRs, the core registers are composed of a certain number of Core Special Function Registers (SFRs) which control the operation of the core and provide status information about the core itself.

The TriCore microcontroller implements Harvard architecture [18], which means physically separate storage and signal pathways for instructions and data. The term originated from the "Harvard Mark I"[8] relay-based computer, which stored instructions on punched tape (24 bits wide) and data in electro-mechanical counters. This architecture is in contrast with von Neumann classical architecture where the CPU can be either reading an instruction or reading/writing data from/to the memory, but one and the other cannot occur at the same

[8] The IBM Automatic Sequence Controlled Calculator was installed at Harvard University in 1944. 51 feet long, 5 tons and incorporates 750,000 parts which was including 72 accumulators and 60 sets of rotary switches, each of which can be used as a constant register, plus card readers, a card punch, paper tape readers, and typewriters.

time since the instructions or data use the same bus system. Then using Harvard architecture permits to concurrent write/read data and reading codes.

The TriCore™ CPU has also a *superscalar architecture*: this means that the CPU executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units (ALU, bit shifter, multipliers, etc.) within it.

Any CPU in the microcontroller accesses separate Program and Data Stretch Pad RAM (PSPR and DSPR), with their own caches and interfaces. This distinction will be relevant while discussing the code/data allocation during the linking phase. While operating context switches[9], a particular attention to the pipelines state will be required, using specific Data-Synchronization instructions (DSYNC) in order to avoid Pipeline hazards[10]. These last are situations that must be avoided. There are 3 types of hazards:

- Structural hazards: when a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute
- Data hazards: when a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available (pipeline stall[11])
- Control hazards: arises from the need to make a decision based on the results of one instruction while others are executing

For more detailed info about hazards and computer structures read intensively [19]. In the TriCore™ CPUs pipeline hazards are minimized by the use of forwarding paths between pipeline stages allowing the results of one instruction to be used by a following instruction as soon as the result becomes available. For the 1.6 Efficiency core, single pipeline architecture is implemented (scalar Harvard), in order to allow a power efficient computation, at the cost of a slower processing.

3.5 Context Management System

An overview of Context Management System will be reported below. A more detailed explanation can be found in the thesis [20] and in the Infineon Patent [21]. In the TriCore architecture, the RTOS layer can be very thin and the hardware can efficiently handle much of the switching between one task and another. At the same time the hardware architecture

[9] Context switch is the process of storing and restoring the state of a process or thread so that execution can be resumed from the same point at a later time. This operation is fundamental for the multi-core system.
[10] Situations in pipelining when the next instruction cannot execute in the following clock cycle
[11] (aka bubble) is a delay in execution of an instruction in an instruction pipeline in order to resolve a hazard

allows a software management of the context switch with relatively few constraints imposed to the system designer by the architecture.

“A task is an independent thread of control”, as defined in [22]. The state of a task is defined by its context. When a task is interrupted, the processor uses that task’s context to re-enable the continued execution of the task when is requested. The lower context registers are similar to global registers in the sense that an interrupt/trap handler or a called function sees the same values that were present in the registers just before the interrupt, trap or call. Any changes made to those registers that are made in the exception routine or call, remains after the return from the event: that means that the lower context registers can be used to pass arguments to called functions and pass return values from those functions.

Contexts, when saved to memory, occupy blocks of storage referred as Context Save Areas (CSAs). The architecture uses linked lists of fixed-size CSAs, and each CSA can thus hold exactly one upper or one lower context, linked together through the Link Word. During a context save operation, the upper and lower contexts can be saved into CSAs. The unused CSAs are linked in the Free Context List (where FCX is the free context pointer) and the CSAs that contain the upper and/or lower context are linked in the Previous Context List (where PCX is the previous context list pointer), as can be seen in [Figure 14].

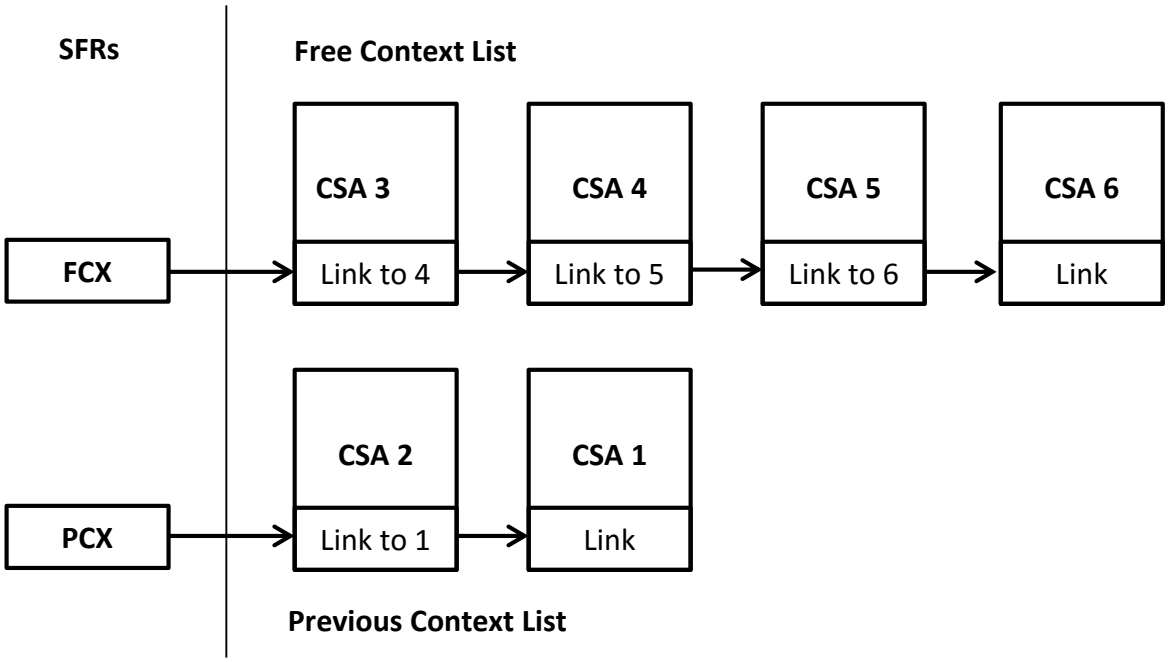


Figure 14: CSA chain before a function call (this image is present on the US 7434222 B2 patent’s extract available online)

The Link Word “Link to 4” e.g. takes account that CSA4 is the next CSA available after CSA3. Before a context is saved in the first available CSA (in this case CSA3), the Link Word of this last is read to supply a new value for the FCX register. PCX is updated as soon as the new task completes its execution. There is also another register that points to the last free CSA: the LCX register. When FCX matches with LCX, the CSA chain is full. An important detail is that the upper context is saved automatically by the hardware instead the lower context is saved through an instruction.

3.6 System Timer and Interrupts Management

Any CPU has a System Timer Module (STM), after a reset is always enabled and starts counting. The timer is implemented as a 64-bits upward counter register running at the system frequency. There are 7 registers (STM_TIM0…STM_TIM6), each with different timing range and resolution. The content of these registers can be compared with other registers (STM_CMP0 and STM_CMP1). Each CMP register has its compare match interrupt request flag that is set by hardware on a match event. An interrupt is an exception signaled by a peripheral or generated by a software request. Not all interrupts has the same priority, for this reason is defined an interrupt priority level for any of it. Service Request Nodes (SRNs) are linked to the Interrupt Control Unit (ICU). This last manages the arbitration among the requests from the SRNs, provides the winner and checks the signal integrity possible errors. SRNs instead are nodes whose skill is to request the interrupts and have registers that takes account of priority level. When an interrupt is triggered by a CPU, an Interrupt Service Routine (ISR) is executed as a callback function[12]. While the processor executes an ISR, it works in an isolated context since an interrupt with higher priority have been triggered. For more info refer to [20].

[12] It is a function that is passed as an argument to another function

4 Operating System

4.1 Why a Multi-core/Multi-task Functional Test Operating System?

In [Paragraph 1.2] has been introduced the road traveled by the company from the old Built-In Self-Test mechanism to the Functional Test Operating System. This version, nowadays used by Infineon's e-Flash test engineers, is a single-core OS[13]. This system permits to test different families of products respect to BIST where there is no portability because any test must be written for any microcontroller and stored on any of it. But, the increase of the requested computing power for the AI[14] has pushed the technology to realize multi-core CPU architectures and increasing the e-Flash dimensions for storing more pre and post-elaborated data. Then, the single-core Functional Test OS has become obsolete, because the test times are increased and the software don't gives total hardware coverage (a single-core OS on a multi-core architecture). So, the MC team proposed to improve the OS with a multi-core version. As it can be seen in [Figure 15], with a single CPU it can be test a memory bank at time (sequential memory test) and if the memory dimension increase the test time increase. With a multi CPU system it can be possible to concurrent test the memory banks, reducing the test time respect to single core architecture. The MC team decided also to have a multi-task OS. Thus, as already said, the target is a multi-core multi-task operating system.

But, let's do some order.
For definition a multi-core OS is a system that can handle a multiprocessor architecture. There are several types of it:

- **Symmetric** Multi-Processor (SMP) system: each core shares the same OS and user applications, resulting in a system in which each core can run all programs and perform all operations. It is reliable, use resources effectively, can balance workloads well but it is the most difficult configuration to implement
- **Loosely Coupled** Multi-Processor system: each core has its own memory, I/O devices and operating system. The cores can communicate each other. When a job arrives, it's attached to the free processor. To keep the system well balanced and to ensure the best use of resources, job scheduling is based on several requirements and policies. When a processor fails, the other continues to work independently

[13] Stands for "Operating System"
[14] Stands for "Artificial Intelligence": the ability of a computer or other machine to perform actions thought to require intelligence. Among these actions are logical deduction and inference, creativity, the ability to make decisions based on past experience or insufficient or conflicting information, and the ability to understand spoken language (definition of "The American Heritage® New Dictionary of Cultural Literacy, Third Edition")

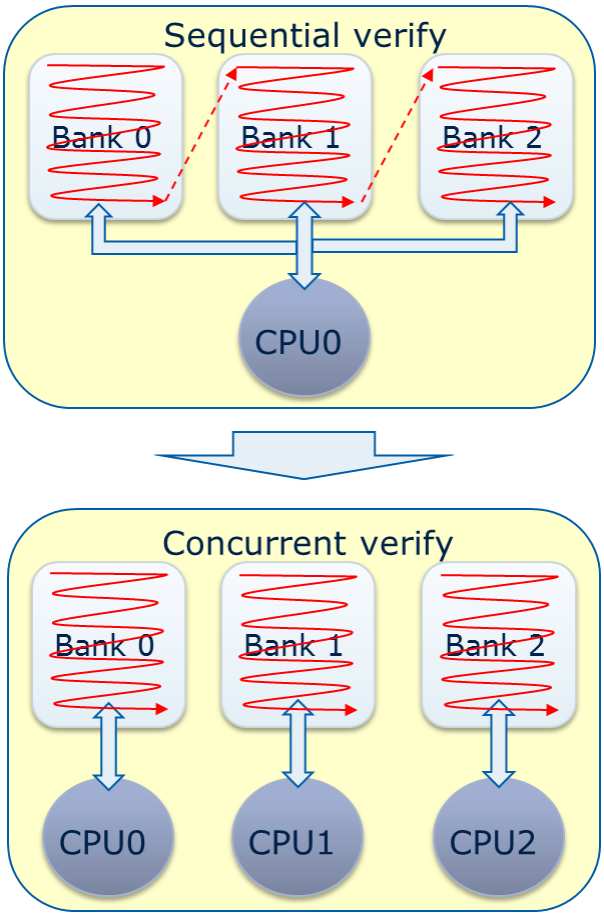


Figure 15: Single-core vs Multi-core e-Flash tests

- **Master-Slave** Multi-Processor system: only one core, referred as the Master Core, has a complete access to all OS code and to all user applications. All other cores (Slave Cores), can access only a small subset of the OS and are usually triggered by the Master Core in order to execute any program. This configuration is well suited for computing environments in which processing time is divided between front-end and back-end processors; in these cases, the front-end processor takes care of the interactive users and quick jobs, and the back-end processor takes care of those with long jobs using the batch mode. The reliability is not higher because if the master core fails, the system fails. It increases the interrupts because anytime slave cores need an OS intervention they must interrupt the master core. It can lead to poor time management because a free slave core must to wait the master core for the next operation

As we see, there are various pro and cons. The TriCore™ has a master-slave configuration, because the SMP solution was discarded due to the HW limitations of the SoC. Another

problem to the SMP configuration was coming from the ATE communication protocol, which expects the executions of flows to be sequential, and not parallel, so ATE commands are always triggered one by one. This also limits the parallelization of the execution if no update to the ATE interface is desired. But it has also a scheduler that permits the multitasking. This means that is not the traditional master-slave configuration.

Multitasking instead is not related to the core architecture, but is related to the processes scheduling. It can be implemented also in a single-core CPU. More rigorously, it is a synonym for multiprogramming, a technique that allows a single processor to process several programs residing simultaneously in main memory and interleaving their execution by overlapping I/O requests with CPU requests.

Thus, the engineers choose to implement a multi-core master and slave OS with a multitasking ability for any CPU. The scheduler was designed and developed during multi-task and also multi-core improve. This not means that the scheduler is designed for having a multi-core OS. The multi-core speeds up the e-Flash tests execution. The multitasking instead improves any core's throughput. This last is the rate at which the processes are executed. In pragmatic way, the multitasking avoids dead time in the mechanism of processes execution on any core. Finally we can answer to the initial question: it is used a multi-core multi-task OS because it is improved the e-Flash tests speed using a master-slave configuration (this to maintain a lower power consumption when only one core is needed to execute) and the multi-task feature needs to compensate the problem of a low throughput of a traditional master-slave configuration. For more theory definitions and deepening of theory read [22] and [23].

Now that the idea is clear, we can proceed to the OS structure. The multi-core version has been coded from the single-core version. Thus, we can analyze some basic things that we have also in the multi-core version. The Functional Test OS is initialized at the startup of the testing sequence (where only master core is running), and is active across the whole execution, in order to provide a standard interface with the ATE and the DUT, handling the correct scheduling of the test flow. The ATE can start the execution of Task Applications from the testware libraries, while the Task Application itself sees in the FTOS a hardware abstraction of the Device Under Test. The list of sequence of test is grouped into some Flow Tables (FT) where any element contains the list of tests that must be performed. Flow tables are loaded in RAM with the TW library, and thanks to the FTOS the ATE can ask the DUT to execute all the tests inside the FT one by one. After the entire sequence of test executed, the OS returns a log of the tests executed. In [Figure 16] are represented the operating system modules and how can be accessed.

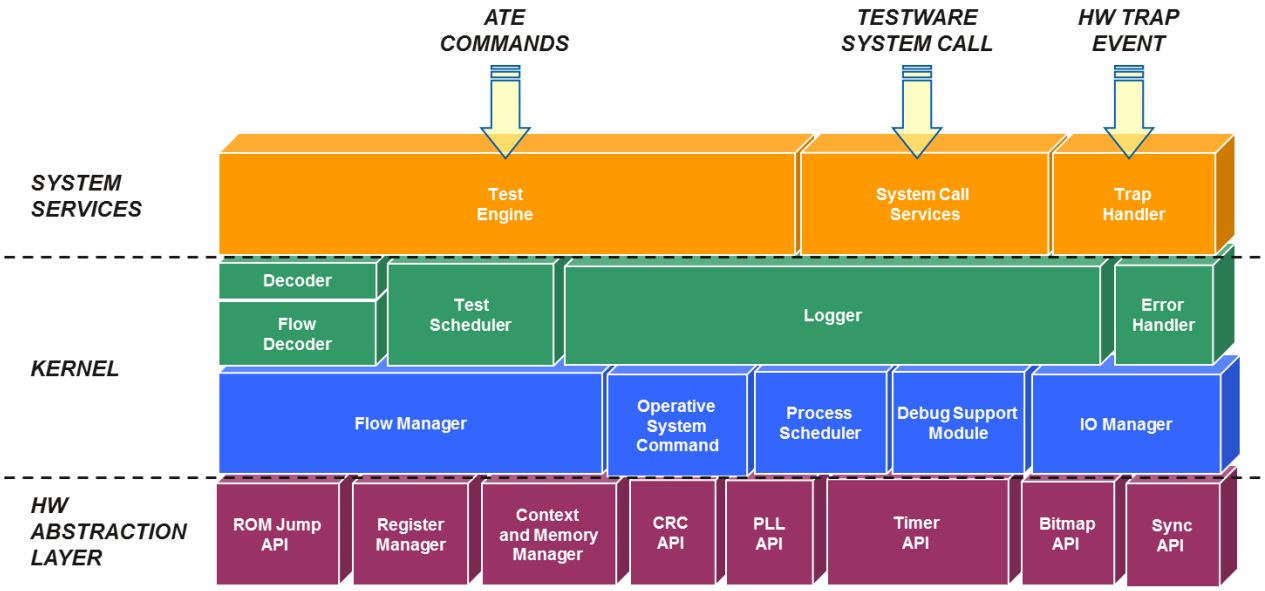


Figure 16: OS modules and layers

Functional Test OS can be accessed through:

- **ATE command interface**: ATE access to the RAM locations and writes the opcode of the command
- **Testware System Call**: when a Task Application (TA) needs to perform a low level operation a jump to FTOS is taken. This allows the upper code layers to be more portable across different microcontrollers (µCs) families
- **Hardware Trap Event**: If an illegal instruction (contained in the trap vector table) is performed, the trap handler provides informations about the anomalous termination

The tests are executed via flow commands. The Flow Manager module permits to embed test flows into DUT and to handle their execution threads. Tests are stored in DUT SRAM in a structured Flow. Flow Manager saves the index of test to run and the flow status.

Regarding the multi-core scenario, the master core is the only CPU running at the startup and awakes the other cores during the initialization sequence. All the commands from the test equipment are received from the master core that coordinates the other cores deciding what processes they must execute. All slave cores are always on waiting for new commands coming from the master and can't create any process. The multi-core feature has been introduced for a specific task called *Verify*. This last contains a list of tests performed of the e-Flash memories.

The new Functional Test OS guarantee the backward compatibility with the old operating system. It also guarantee the scalability of the system to more CPUs, more e-Flash banks, major dimensions of Word Lines, Sectors, Pages, redundancy and so on. For more detailed informations see [20].

4.2 Scheduler Module

The scheduler is the module that choices what process to run next. To describe the OS scheduler, we must to define and explore what are processes and the difference between threads. A process is all the software that is runnable. There are two types: foreground processes (principal processes that interact with the user) and daemons (processes that run in background). A process could be in one of the next three states:

- **Running**: when a CPU is using this process
- **Ready**: when the process execution is stopped temporarily till another process, usually with high priority, is running on the same CPU
- **Blocked**: unable to run until some external event happens

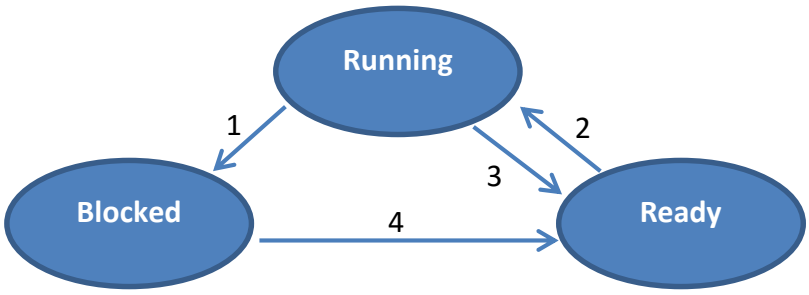


Figure 17: States possible transitions

Four transitions are possible:

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

A process is an entry of the process table where are stored informations, stack pointers, program counters, memory allocations, etc. about the process. For any process there are lightweight processes called threads. When a process thread interrupts its execution, another thread of the same process continues to run. This multitasking solution permits a high throughput, because the core has no dead time. The [22] is a good reference for what concerning OS, processes, threads and schedulers theory.

In the new FTOS, the scheduling policy is a *round robin* which accepts a configurable maximum number of processes, priority levels and maximum call depth per process. To each process is assigned a time interval, called its quantum, during which it is allowed to run. If a process is still running when its quantum is finished, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks. All the schedulers need to maintain a list of runnable processes. An interesting issue is the length of quantum. Switching from a process to another requires some time for doing the administration (saving and loading registers, updating tables, flushing and reloading caches, etc.). This operation is called *context switch* (see what has almost been said in [Paragraph 3.5]). If the quantum is set too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests. The scheduler has a *priority scheduling*: each process is assigned a priority and the runnable processes with highest priority are allowed to run before which have a lower prio[15]. All other processes are organized in queues [Figure 18], sorted by the level of priority. When no process is running, they are all listed inside a "Free processes queue". When a new task is needed by the CPU, a process for that task is created, popping it from the list of free processes. When the process is created, a priority level is assigned to it, and it is immediately pushed inside the proper priority queue. If no other process is ready for the execution, or if the process wins the priority competition, that process is set as running process for a quantum time slice. During the quantum time slice, the process can't be interrupted by the scheduler, and it continues its execution until next scheduler's step, which is automatically triggered by a System Timer Interrupt. The duration of the quantum is configured during the initialization procedure of the scheduler itself, and when the time quantum elapses, the process' context is saved and the scheduling routine starts executing, also if no other process is ready. Instead, if new processes are available, the schedulers set as running the ready process with the highest priority, keeping other processes in wait until the higher processes terminate their execution. Round robin policy is applied for processes which share the same priority level [Figure 19].

[15] Abbreviation of "priority"

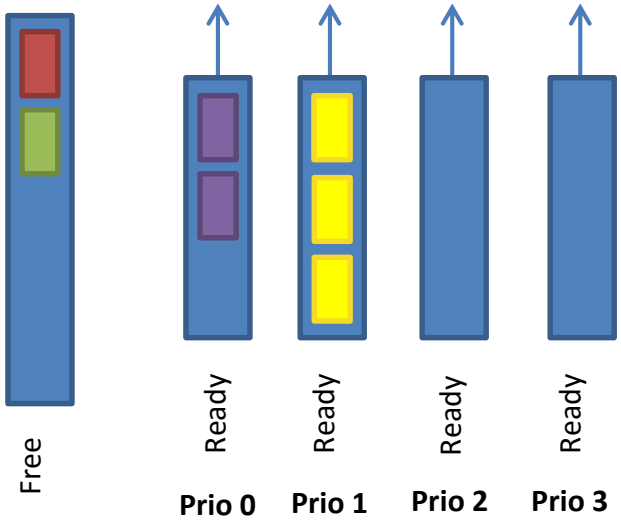


Figure 18: Free process queue and priority queues

The processes are implemented as structures in C programming language. They contains the context registers (LCX, PCX and FCX) and the pointer to the users stack (pstack). A pointer to the next_process implements the process queue. The process_id and the index are useful to identify the process among the whole program (process_id) and inside its queue (index). For more detailed info on the scheduler implementation refer to [20].

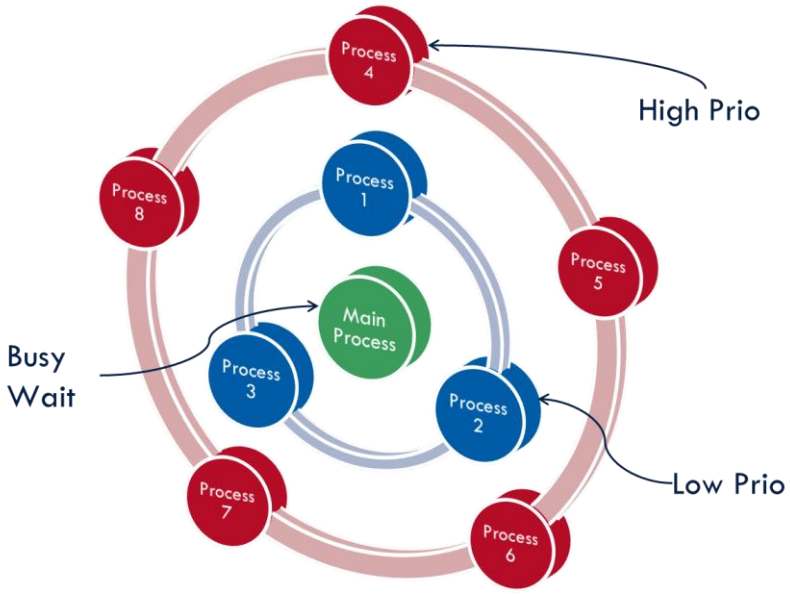


Figure 19: Round robin scheduling

4.3 Multi-core Verify and Iterator Module

As it can be seen in [Figure 15] some pages back, the multi-core verify permits to reduce the test time used for testing memory banks using a concurrent e-Flash modules scanning. The function flowchart is depicted below in [Figure 20].

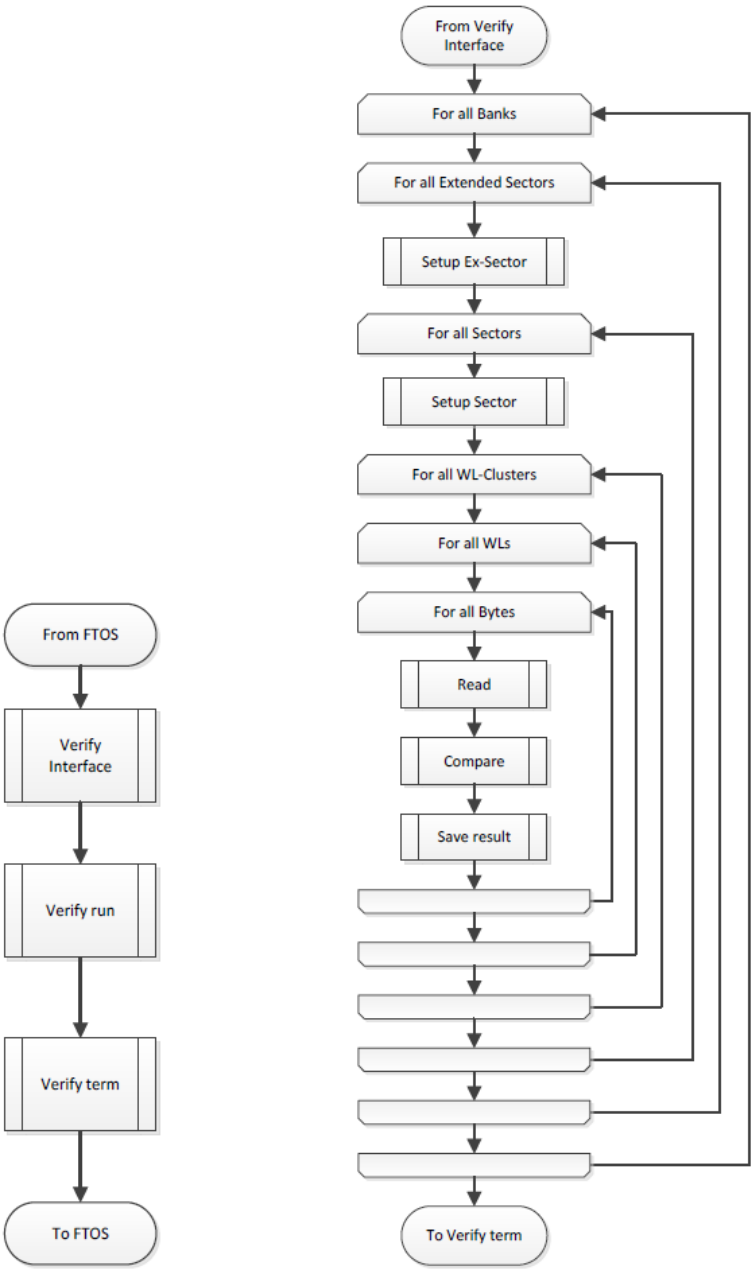


Figure 20: Verify function flowchart

The *Verify Interface* decodes input parameters from the RAM and consequently initializes input and output structures to be used inside the *Verify* test. The *Verify run* is responsible for the execution of the test. The *Verify term* is the function which merges all partial results and passes the relevant ones to the FTOS for the final logging. The *Verify* run is stratified in the same manner as the e-Flash memory:

1. Bank
2. Extended Sector
3. Sector
4. Word Line Cluster
5. Word Line
6. Page
7. Byte
8. Bit

Inside of any memory layer occur a not-priori defined number of the *Verify* iterations. The iterations are test performed repeatedly over the e-Flash memory like 0s, 1s, checkerboard, etc. As *Verify* outputs are logged: data errors, redundancy errors, unrepaired failing bits, pump load errors, timeouts and the debug table.

The iterations are controlled by an *iterator module* which is part of the test library, and not specifically related to the *Verify* functions. The iterator is the SW module which performs the *Verify* cycles for each section of the flash. The multi-core *Verify* has some synchronization functions that permits to align all cores at the same iteration level, after all the CPUs has finished to iterate over its own memory section. The operations performed are the same for all iteration levels. The synchronization functions are inside the iteration module, which in the new version of FTOS will be responsible for the alignment of all cores. Each time a memory section has been verified, the Special Function Register (SFR) configuration is changed for all cores by one core (not necessary the master core). The synchronization for levels below the sector iteration is not required. Across different banks there are different number of extended sectors, sectors, WL and so on, while all cores recognize what is the iteration level of other cores in order to understand if they have to wait or not for them. In [] is reported the multi-core update for the iteration module: this structure has some issues and takes part of the modification targets of this thesis.

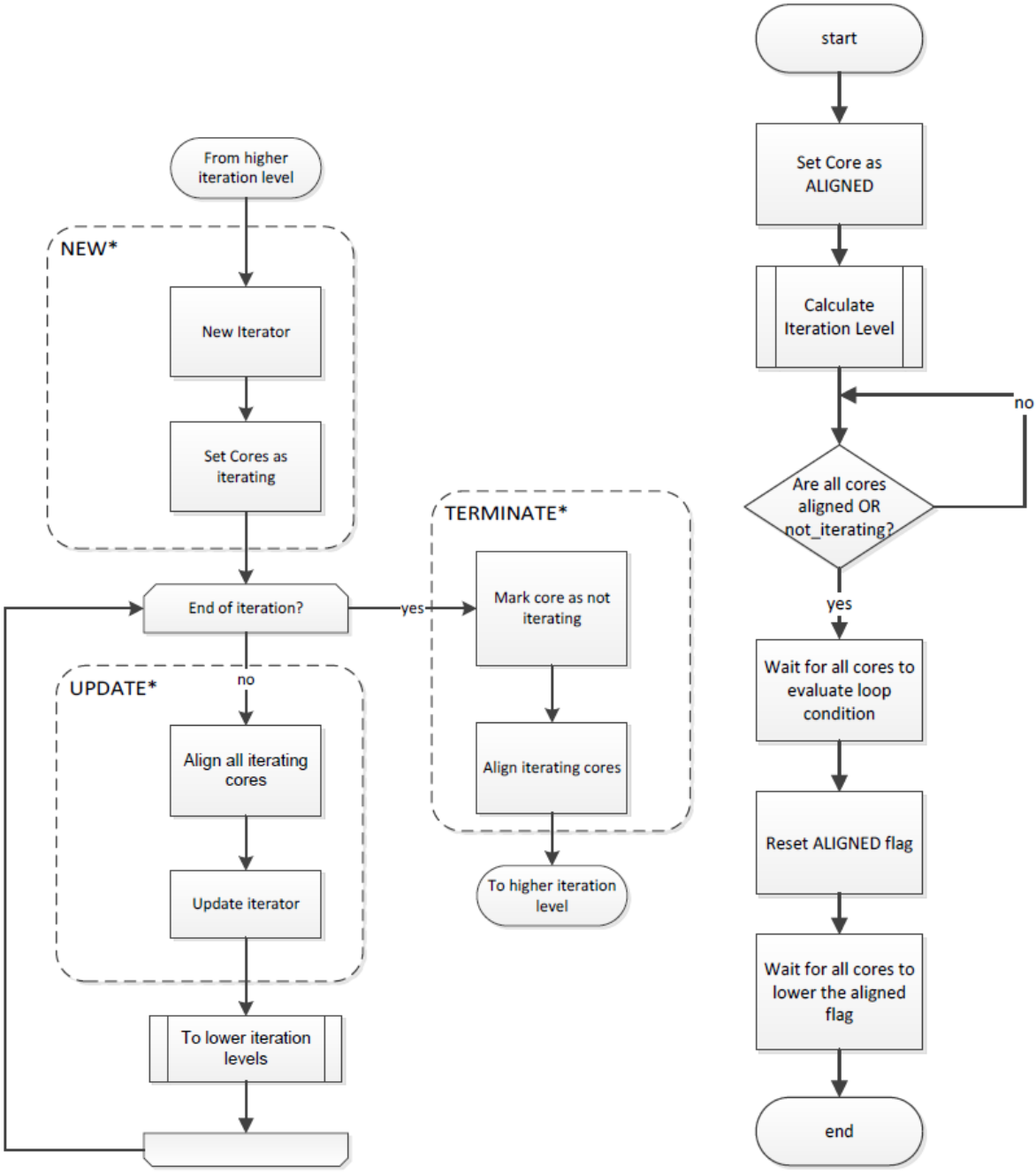


Figure 21: Iterator flowchart

The multi-core iterator has Look Up Table (LUT) that defines what memory sections are tested by any CPU. The LUT is writable in RAM before the ATE triggers.

LUT	Bank 0	Bank 1	Bank 2	Bank 3	Bank 4
CPU0	1	0	0	0	1
CPU1	0	1	0	0	0
CPU2	0	0	1	1	0

Figure 22: Example of LUT, before iteration

If the master core LUT row has all zeros (CPU0 has no banks assigned). The slave cores must wait the start command by the master. To avoid this problem, in the new FTOS, has been introduced a new status flag: the *promoted core*. This allows the master core to choose a slave core and give it the executions of functions that normally execute only the master. The promoted core is able to:

- Access to Special Function Register for the synchronization
- Control the master core's alignment flag

4.4 Test Results Merging

Once the LUT has been emptied from all bank *Verify* requests, the results are still contained in the temporary variables of each core, and they have to be merged together. Indeed the higher software layers do not expect to receive N results for N CPUs, because in their eyes the *Verify* function is still executing in a sequential mode. Thus output results have to be merged. Most of results are cumulative, i.e. they represent the sum of all errors inside a bank, or the number of repaired bits, number of failing 1s, number of failing 0s and so on. Only the sum of their values is relevant. The flowchart of how the merging is implemented is reported in [Figure 23].

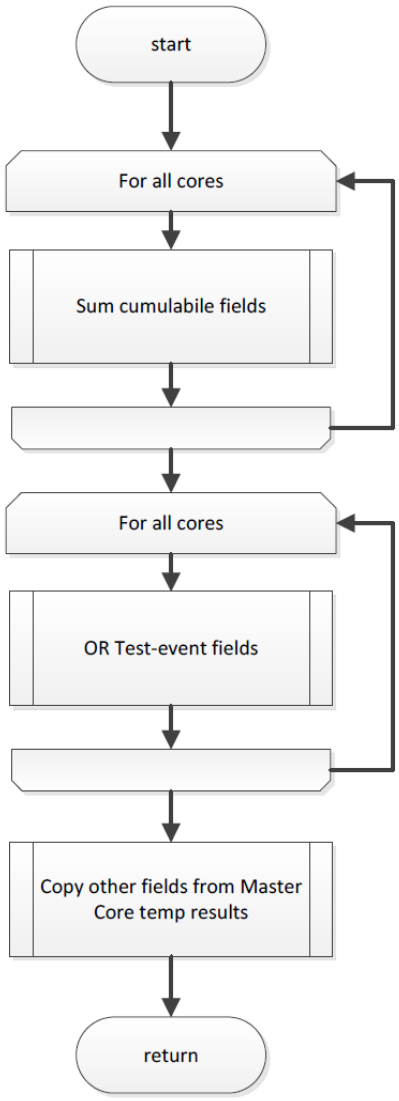


Figure 23: Multi-core verify results merging flowchart

5 Operating System Release

Any developed software follows a Software Development Life Cycle (SDLC) shown in [Figure 24]. When this life cycle is completed, a new phase called *release period* begins. Also for this last, is defined a standard Software Release Life Cycle (SRLC) that allows standardizing, in a more generic and flexible way as possible, the phases of software release. The major stages of software release are:

- **Pre – Alpha**: Software doesn't necessarily contain completed features/functions. It is often an interim product build, prior to testing, often to validate pieces of work or that development to this point hasn't broken the build process. Examples of activities are: requirements analysis, software design, software development, unit testing and *nightly builds*[16].

- **Alpha**: It is the first phase to begin software testing usually with the white-box approach[17]. Then, grey and black-box tests are performed. It is often a preliminary build that is only partially complete and typically contains temporary codes, comments, product breaks, etc. Alpha software can be unstable and could cause crashes or data loss. The α-phase usually ends with a "feature freeze" that means that no features will be added to the software.

- **Beta**: On this phase the software is the first version released outside for the real-world testing, must include all the features but often contains known bugs. The idea is to introduce the beta version in the market or to the costumers in order to have available a huge number of testing users that give feedback about the product and its issues increasing the probability of detecting faults.

[16] The term is frequently used for large projects where a complete rebuild of the finished product from source takes too long for the individual developer to do this as a part of their normal development cycle. Instead a complete rebuild is done automatically during the night so the build computer have 8-10-12 hours to do the build and have it ready for the developers coming in the next morning, so they can continue working on their individual tiny bit on top of the new version.

[17] White-box testing is associated with source code testing (i.e., unit testing). For example, correct infinite loops, unreachable code problems or undefined variables. Black-box testing instead, ensures that those parts of the applications that will be exposed to the user work correctly (i.e. check that the requirements are reached). Finally, the grey-box approach is a hybrid version of white and black-box approaches. More info at [34].

- **Release Candidate (RC)**: It has all the features and has the potential to be a final product but there may still be changes to fix defects.

- **Release To Manufacturing (RTM)**: It has the peculiarities of being stable and as much as possible bug-free because has been bug fixed. It only states that the quality is sufficient for mass distribution.

- **General Availability Release (Gold)**: It is the official final version of the product that will be released to the public.

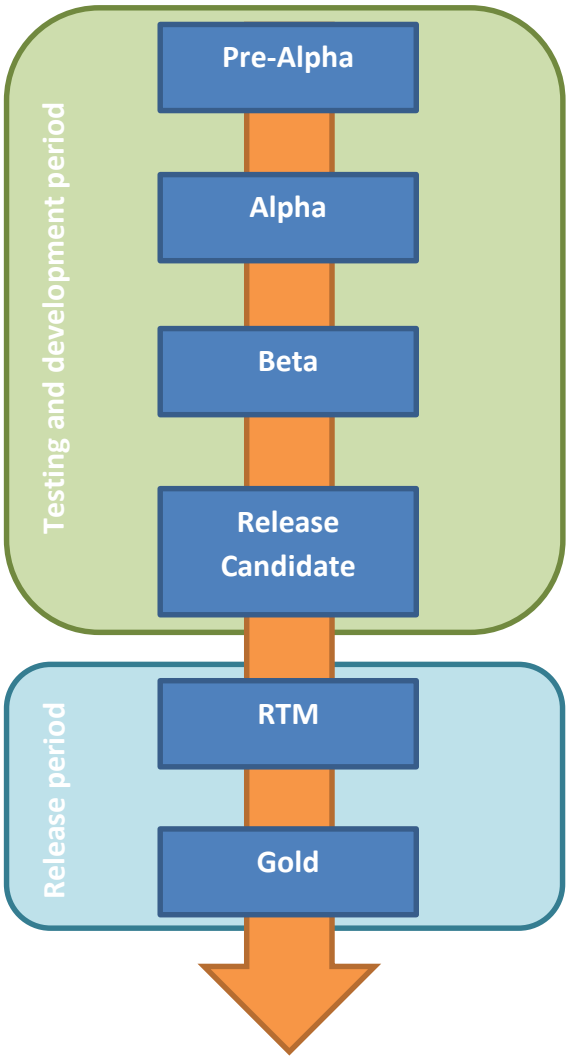


Figure 24: Software Release Life Cycle

One of the aims of this thesis is to analyze the process of issue fixing to permit the β release of FTOS multi core version. The FTOS release, nota bene, stops at the RC. To allow the release of the multi-core operating system, the functions related to the cores

synchronization were changed and we done a feasibility study to figure out how to clone the code in the CPUs RAM memories. Following the feasibility study, has been chosen the best solution according to requirements and the time available to release the FTOS.

5.1 Synchronization

In computer science, synchronization is an enforcing mechanism used to coordinate and manage processes execution and manage shared data [24]. This general definition is valid for processors, microcontrollers, computer networks, etc. The *data synchronization* refers to the idea of two or more processes that have to access to the same data in memory. If there isn't a mechanism that coordinates the access to this resource, it could be corrupted. The *process synchronization* instead, permits to avoid that two or more processes simultaneously execute the same critical section[18] or to align processes with different execution time that have been executes in parallel.

5.1.1 Mutex

The Mutual Exclusion (*mutex*) is a type of data synchronization mechanism where a process blocks a critical section (in this case a shared memory data) and one or more processes wait to get their turn to read/write/copy the data. What could happen if we haven't a mutex mechanism for accessing shared memory data? Ideally, if two processes arrive to the data they access together resulting in a conflict. But, in reality, the complexity of the operating system due to the scheduler and the very slightly different access times to the data in memory make impossible two processes to access at the same time to the same memory location. This implies that one *process A* accessing to the data, e.g. for writing it, and a *process B* accessing slightly after *A*, e.g. for reading, results in a data corruption because who is reading could read the data modified by who is writing [Figure 25]. With the mutex the *process A* access to the data and holds the token until it finishes the operation. While the token is held by *A*, the other processes busy waits. Then, one of them can access to the unlocked data [Figure 26].

[18] It is a part of a program that must not be executed concurrently by more than one processes

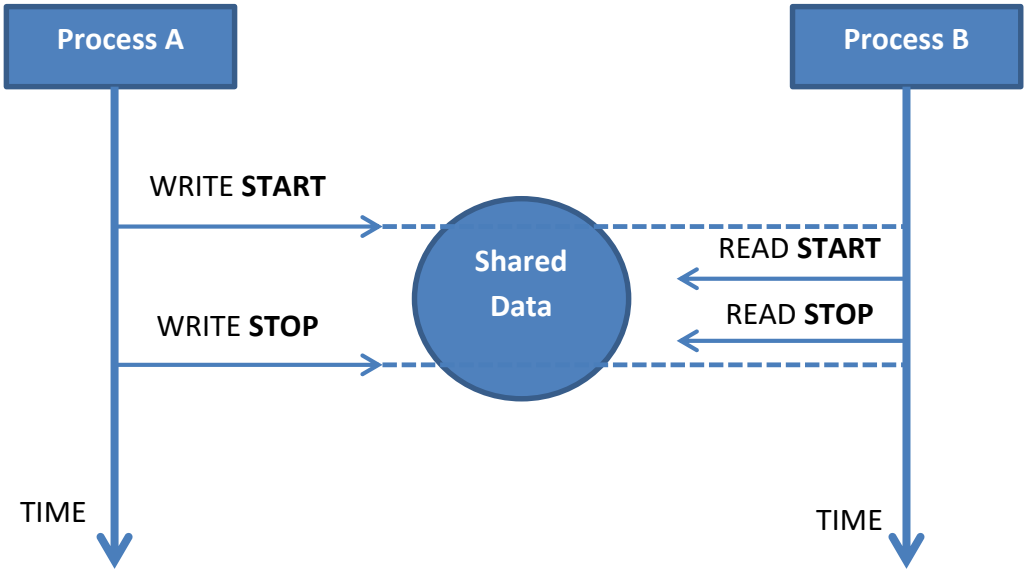


Figure 25: Two processes accessing to the same data without a mutex

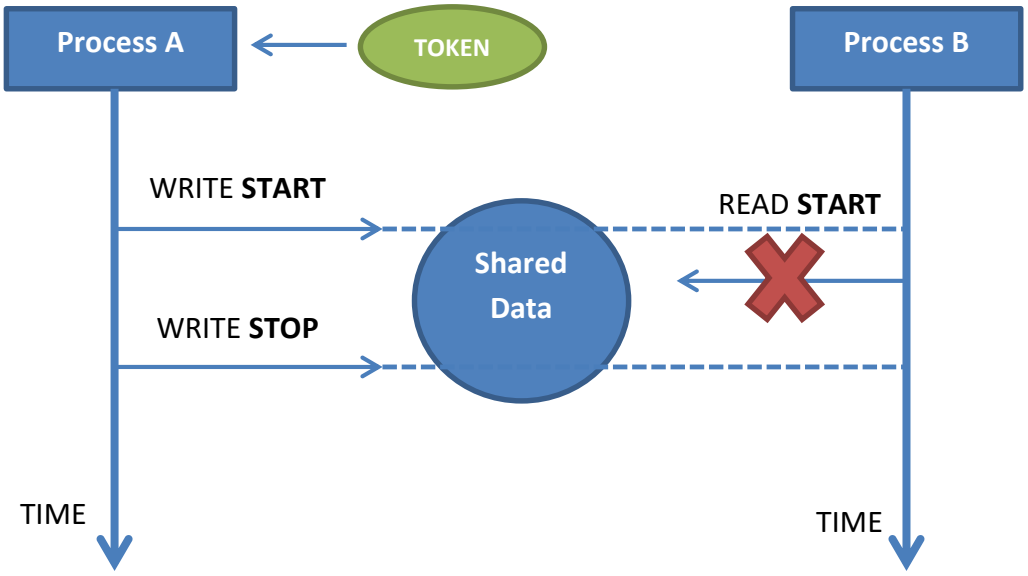


Figure 26: Two processes accessing to the same data with mutex. Process A has the token, process B busy waits until its turn

E.W. Dijkstra first identified the requirement of mutual exclusion in 1965, in his paper [25]. FTOS controls the mutual exclusion and depends on the scheduler granularity. There are many types of mutex solutions:

- **Locks**: each process cooperates by acquiring the lock before accessing the corresponding data. A lock allows only one thread to enter the part that's locked and the lock is not shared with any other processes)
- **Semaphores**: same as locks but allows a predefined number of processes to enter
- **Monitors**: lock + condition variable. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task
- **Message passing**: exchange messages between processes (used in object oriented operating systems)
- **Tuple space**: a tuple is a finite ordered list of elements. Processors post their data as tuples in the space and the consumers processors retrieve data from the space that match a certain pattern

For our purpose, where we code in C, we choose to implement the simplest type: the locks.

5.1.1.1 Mutex Requirements and Concept

Data synchronization was introduced into the multi-core FTOS to handle 3 CPUs accessing to shared registers and shared data. When registers or data are shared between conflicts could occur. The first implementation of data synchronization is reported in [Figure 27].

1. When a CPU want to access to a shared data locks a public[19] variable
2. Checks if other cores are locking
3. If the answer is yes, the CPU unlock the public variable and waits for some cycles to relock the public variable
4. If the answer is no, all works well and the mutex section can start

[19] Public because its state must be visible and modifiable by all CPUs

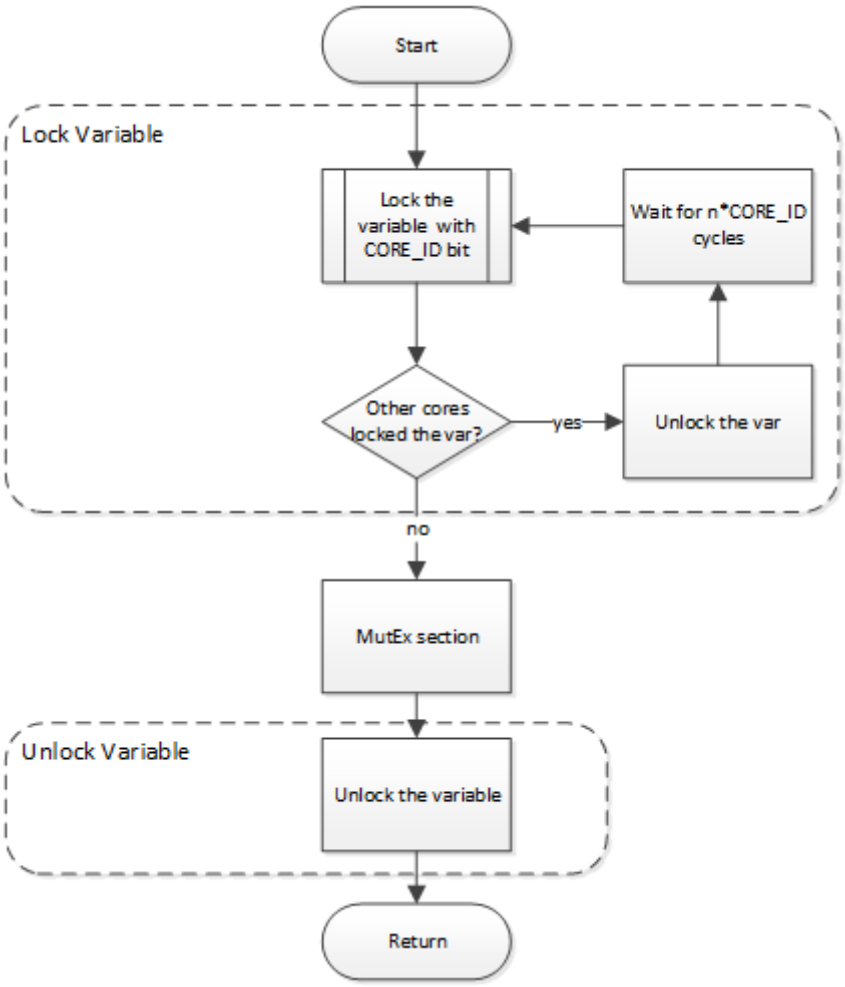


Figure 27: First concept of lock/unlock

This structure (see in [20]) has been improved to have the fastest implementation for the lock/unlock operations. According to the following REQs:

1. Sync function code must be duplicated to each CPUs' RAM to be fast. Lock and unlock functions must be cloned in each CPU because each CPU writes in shared registers (e.g. for errors during the redundancy check during the *Verify* function).
2. All current lock/unlock functions must be replaced with the newer fastest one
3. The lock /unlock code must be implemented with atomic operations to avoid interruptions when interrupts are enabled and possible data corruption or deadlocks.
4. The token, named *var*, must have initial value = 0 and must be a global variable because each CPU could access to this variable
5. the implementation must follow this PSEUDO code

```
INIT
var = 0; \*the token*\

LOCK
reg = 1

do {
        SWAP(var,reg)

   } while (reg != 0)

UNLOCK

var = 0;
```

As we can see in [Figure 28] the flowchart covers the pseudo code. A function is defined for initializing all the global variables needed for the operating system synchronization and the global variable `var` is initially zero. The pseudo code and the flowchart illustrates that:

1. When the i-th CPU access to the lock function, sets the variable `reg = 1`
2. The `SWAP()` operation is atomic: this means that this is uninterruptable from the operating system interrupts. The `SWAP(a,b)` performs an assembly operation where the a content is loaded to b and the b content is loaded to a
3. If the token var = 1 (another core is locking the shared data), the swap operation returns var = 1, reg = 1
4. Loops until var = 0 (no CPU is locking the shared data) and lock function returns
5. When the i-th CPU finishes to accessing to the data (mutex session) the unlock function is called and simply reset var to 0 with another atomic operation.

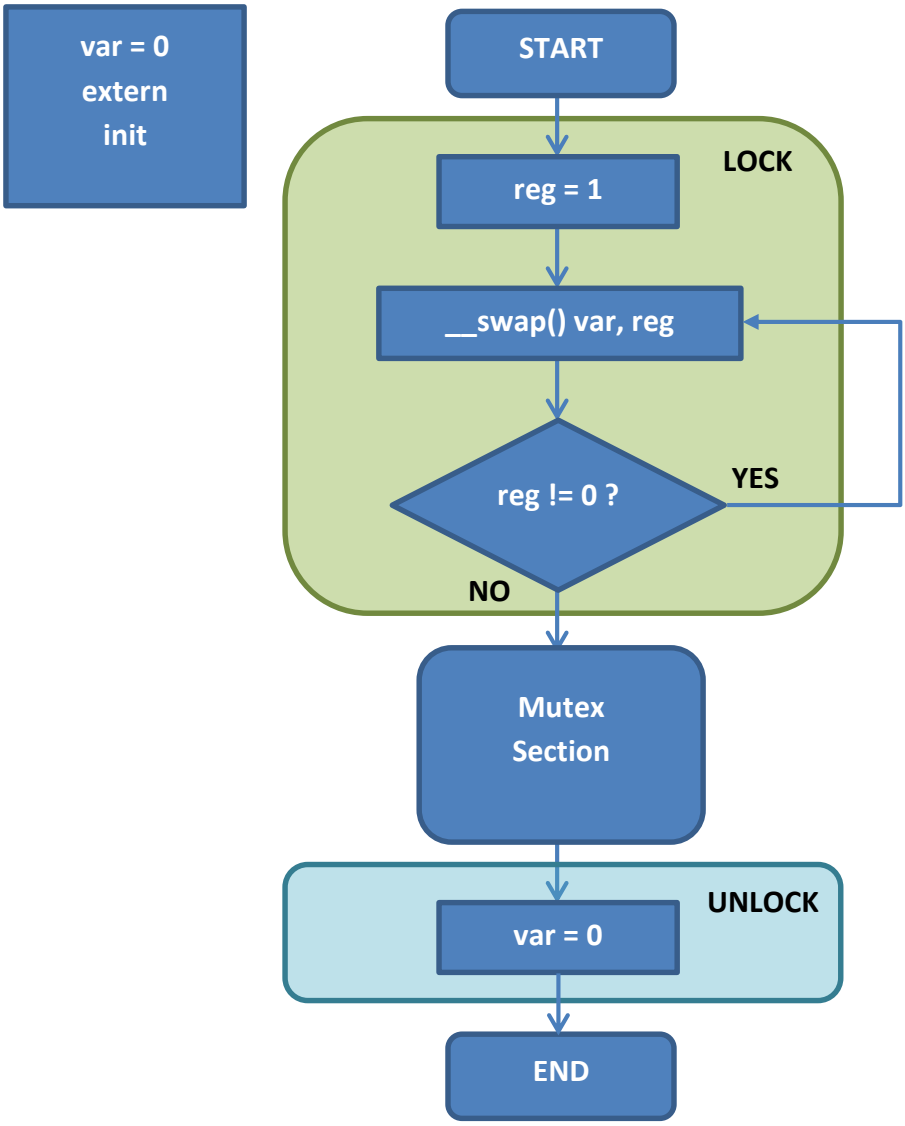


Figure 28: New concept of data synchronization

The differences, compared to the previous concept, are that uses an atomic function and is not a spinlock. This means to strip down the loop that waits for the unlock operation by another CPU. The faster the operations of locking and unlocking are, the better the performance of the operating system are since the operations that access the shared registers are many and will be more and more in future versions of the operating system which, for now, is only in the initial version.

5.1.1.2 Mutex Implementation

The global variable (the token) is initialized in an initializing function. The lock and unlock operations are performed in two separate functions. The reason is that we can lock the shared variable for a slice of code and, after its execution, we can unlock it. Let know that volatile keyword forces the compiler to consider `reg` as a global register that, for any cycle, has to be evaluated and updated without any optimization. We must insert the `SWAP()` within a while evaluation argument, like in [Figure 30]. In this case, the volatile command takes effect because, for any cycle, the while argument is mandatorily evaluated and updated.

```
LOCK_VARIABLE ()
reg = 1

do {
        SWAP(var,reg)

   } while (reg != 0)
```

Figure 29: First lock implementation pseudocode

```
LOCK_VARIABLE ()
volatile reg = 1

while (SWAP(var,reg) != 0)
```

Figure 30: Final lock implementation pseudocode

For the unlock function the solution is simply reset the global shared variable.

```
UNLOCK_VARIABLE ()

var == 0
```

Figure 31: Final unlock implementation pseudocode

In FTOS, for example, the lock / unlock mechanism is used to set the registers of the bitmap and the errors registers without conflicts.

5.1.1.3 *Mutex Verification*

The step of verification and validation ensure that the software reflects the requirements and which meets them in the right way. Namely, the verification is to establish that the software meets the requirements and specifications, so for example that there are no missing requirements, while the validation is used to determine whether the requirements and specifications are also respected in the right way. In our case, we proceed with the verification to ensure that all the requirements assumed are met. To verify lock/unlock functions, the UDE PLS debugger has been used. The microcontroller is linked on a test board and connected to the personal computer by a JTAG connection. This apparatus allows to systematic debug the functions with a step-by-step execution and halting/running any CPU. The UDE debugger allows also seeing the assembly version of the C code compiled.

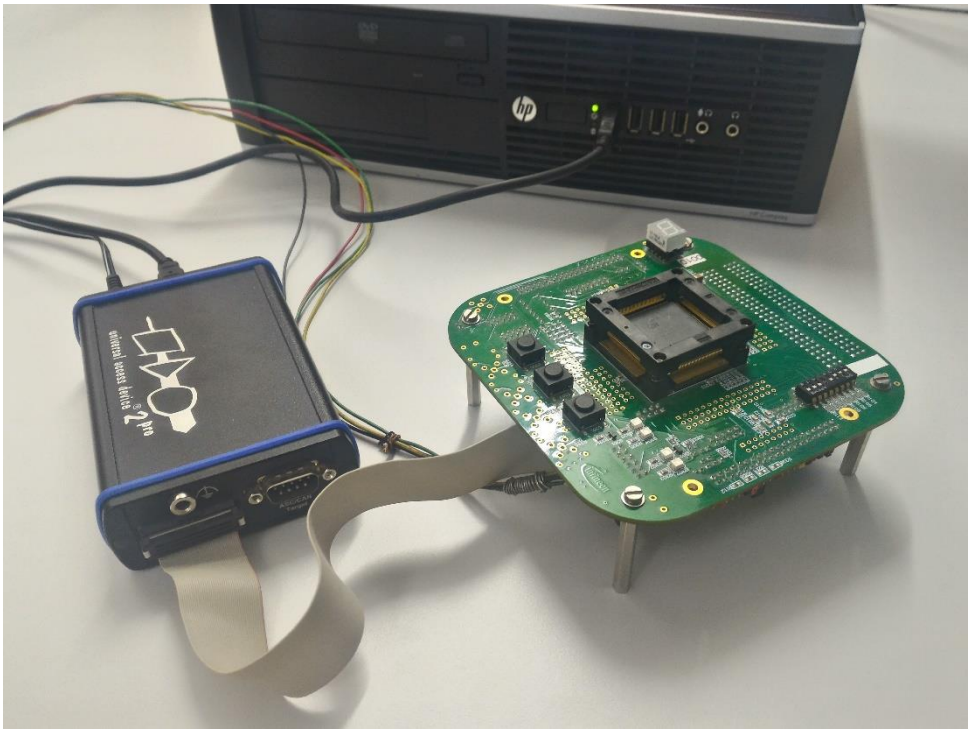


Figure 32: UDE Debugger setup

An initial lock implementation, with the do-while cycle and w/o volatile keyword, did not work properly. Starting from the condition of unlocked shared data (var = 0) and reg = 1, the first swap is done by the atomic operation but, when the execution reaches the while argument evaluation, the reg variable (equal to zero after the swap) is set to 1. The motivation is that the reg variable should be marked as volatile and the swap must be put in the argument of a while cycle instead of a do-while. The *volatile* keyword is a C type modifier and makes sure that what you are reading out of the variable isn't based on the compiler optimization or an old copy of the variable that your program had. The volatile keyword ensures that the variable is fetched from memory on every access, avoiding no-sense values due to the discontinuous variable refresh. In our case, the reg variable is accessed from different CPUs. Thus, the motivation of using this keyword is to prevent the compiler from unwittingly caching values used by multiple CPUs at once [26].

The basic idea of *putting the swap in the argument* of the while cycle is to force the evaluation, at each cycle, of reg variable. In fact, as already said, the swap operation returns the value of reg. Then, the code is different from the pseudo code. This means that this last requirement wasn’t right and should be modified.

5.1.2 Alignment Function

Another type of synchronization function is the alignment function. This last is aimed at aligning all iterating cores to allow changings in the shared registers, for the next tests that have to be executed. The registers are the same for all cores. If two cores are running and one of them finishes the test before the other [Figure 33], it cannot modify the register because its change in the register is reflected in the other core while is running. For example, if CPU1 has finished and sets new currents margins, these new margins are changed also for CPU2 though is running. This case produces test errors or wrong execution tests for the running CPU.

5.1.2.1 Alignment Function Requirements

The old implementation had three alignment functions:

1. *Align All Cores*
2. *Align To Master*
3. *Align Master*

All the alignment functions handle an array of alignment flags. When a core needs to align to each other, a check is done on that array of flags and they continue polling on that flag until not all desired cores have raised their own one. In order to avoid race conditions, a delay is introduced before and after the lowering of the alignment flag. This is the simple implementation of the *Align All Cores*, the first flowchart from left, in [Figure 34]. The wait blocks are the delay just mentioned. The flowchart is valid for all the cores.

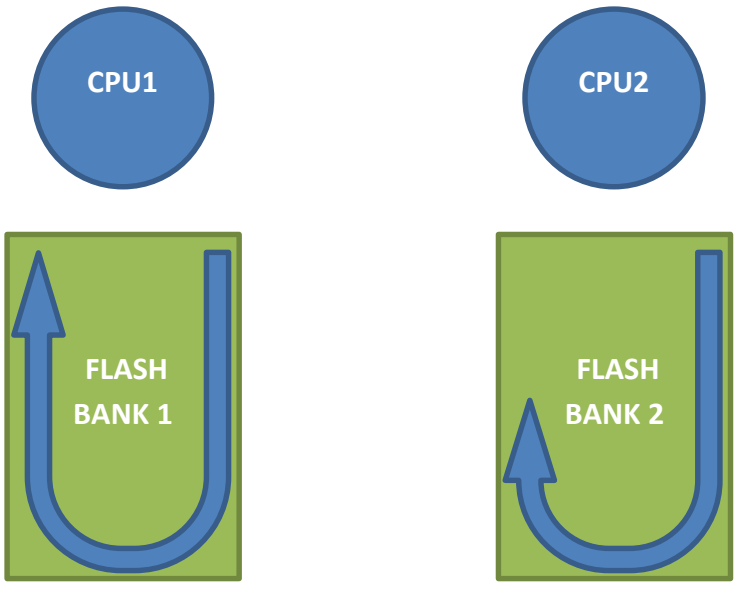


Figure 33: Two cores running the same tests in parallel over two memory banks, one of them ends before

The *Align To Master* is used to prevent slave cores to start executing a function before the master core CPU0 grants its permission, raising its alignment flag. If CPU0 accesses to this function, terminates immediately the execution. If the other cores access, instead, they waits by polling if the master core is aligned. Also there, there is a delay block to wait the master core to lower its aligned flag. All these things are visible on the second flowchart from the left in [Figure 34].

The *Align Master* function is used to raise the alignment flag of the master core for some CPU cycles and it is used in combination with the *Align To Master* function, in order to release the slave cores. In fact, if a core that is not CPU0 access to this function it terminates immediately the execution. Only the master core can access to it and set itself as aligned, wait some cycles and reset itself as not aligned. See the third flowchart in [Figure 34].

While the application using FTOS is executing a verification over the e-Flash memory, only one core (the master core or a promoted core) should perform Special Function Registers operations. The other cores, simply busy waits who is changing the SFR parameters. For this reasons we need the alignment functions to coordinate the halt/run of CPUs while one of them is operating for the others.

However, we have three functions that do similar things. Why do not merge the three functions into one? Now, the multi-core operating system has only one process called *Verify*. This process does several tests over the e-Flash memory using one or more CPU. Each test is done starting from the highest level of e-Flash (what is called bank) and then going down level (extended sectors, sectors, word line clusters ...). For any level, the test is iterated. The cores that are running the test are defined as iterating cores.

We propose a new Alignment function based on the following requirements:

1. There might be a unique and shared among all available CPUs synchronization function
2. Function code must be duplicated for each CPU
3. Only iterating cores must be synchronized
4. All current alignment functions must be replaced with the newer one
5. The implementation must follow the scheme in [Figure 35]

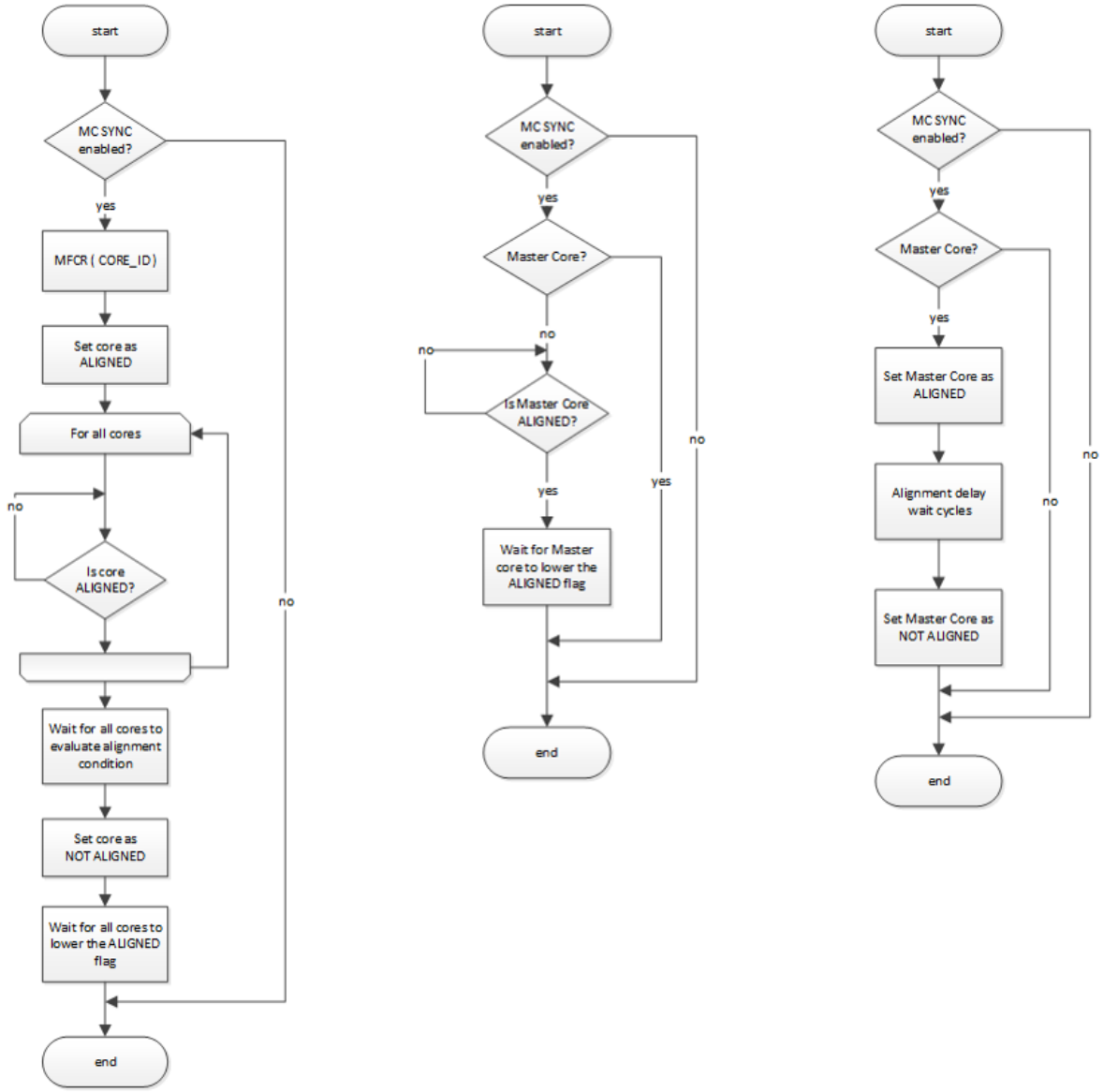


Figure 34: Old alignment functions flowcharts

The scheme in [Figure 35] follows the following steps:

1. Core i-th calls the ALIGN and sets i-th bits in the others *Sync Pointer*
2. Waits until its masked *Sync Pointer* is full of zeros
3. Resets its *Sync Pointer*

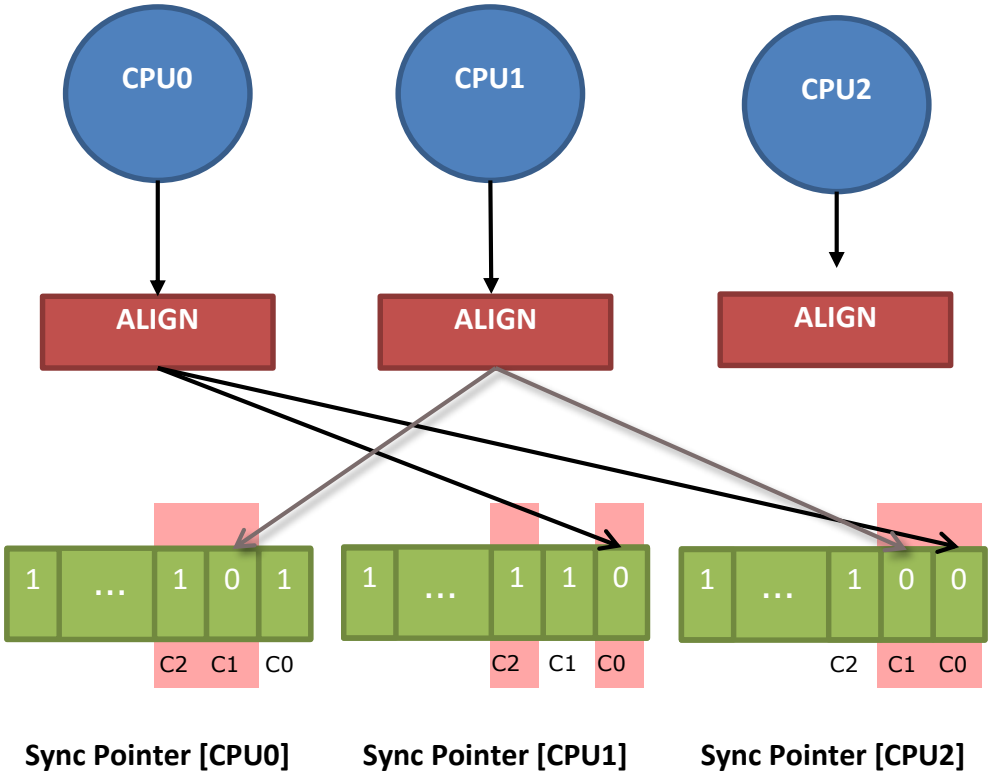


Figure 35: Alignment function scheme

The motivation is to make more robust and reliable the alignment between all CPU with a unique complex function with the purpose to avoid at all (or limit to the minimum) possible blocks/deadlocks.

5.1.2.2 Alignment Function Concept

A unique alignment function must be realized. Each CPU has its Scratchpad RAM that is a high-speed internal memory used for temporary storage of calculations and data. The SRAM has Harvard architecture: this means that it is divided into data SRAM (DSPR) and program SRAM (PSPR). Thus, the i-nth CPU has its DSPRi and PSPRi. The idea is to realize a variable structure usable in future with more than three CPUs, setting a parameter in PSPR that defines the number of CPUs. In our case, now, is set to 3. The concept is reported in [Figure 36] and treats the case with three cores. The blue circles are the CPUs, the green blocks represents the memory location where the alignment flags are written, the orange squares are the data SRAMs and the blue lines represents CPU1 that is writing on the others SRAMs its alignment flags. This mechanism is the same for all the CPUs: any CPU accessing to the *Align All Cores* function, first, writes its aligning bits.

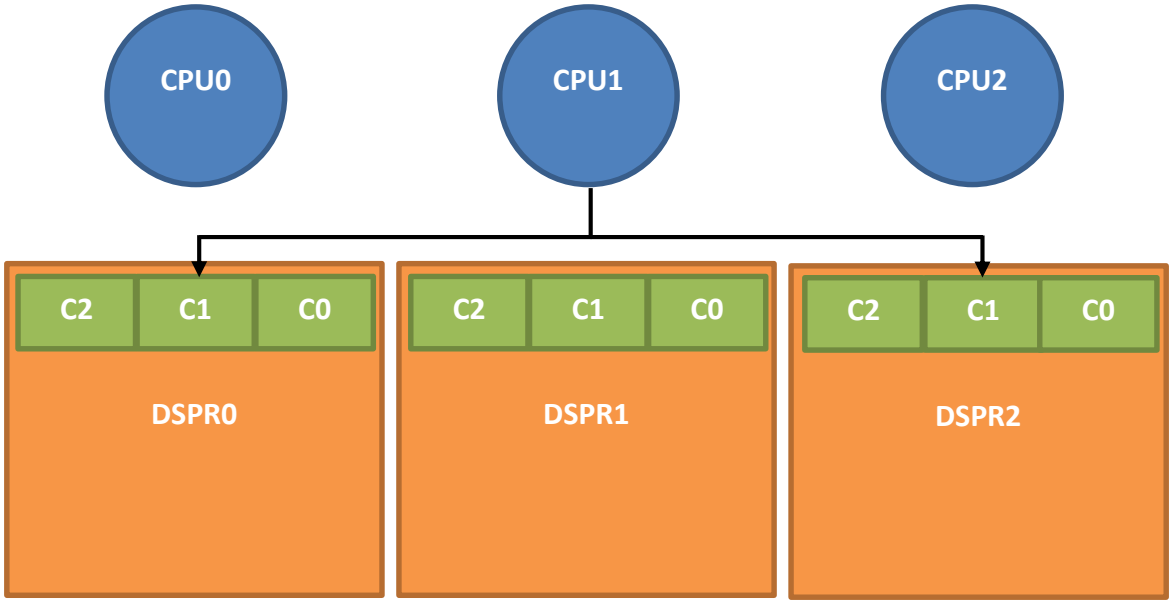


Figure 36: CPU1 writing the aligning bit to the others CPU's SRAM locations

Then any CPU waits until all the other iterating cores are aligned. When this condition is checked, each CPU involved leave the waiting cycle and resets its aligning variable in its DSPR. Therefore, for the aligning bit we define, in our case, three variables with 1 bit for any CPU. The bits are grouped as a vector defined as *Sync Pointer* where the least significant bits (LSBs) are:

$$\{bit_2, bit_1, bit_0\} = \{C2, C1, C0\}$$

and bit_i is set by CPU_i with $i = 0, 1, 2$. Moreover, any *Sync Pointer* $\{bit_2, bit_1, bit_0\}$ is defined for the y-nth DSPR for the correspondent CPU_Y. The bits are defined as

$$bit_i = 1 \rightarrow CPU_i \ \ NOT\ ALIGNED$$
$$bit_i = 0 \rightarrow CPU_i \ \ ALIGNED$$

For example:

$$[0,0,0] \rightarrow all\ cores\ ALIGNED$$
$$[1,1,1] \rightarrow all\ cores\ NOT\ ALIGNED$$
$$[0,0,1] \rightarrow CPU0 \ \ NOT\ ALIGNED$$

To select the bits that must be set for any CPU involved, a mask is used:

$$1_B \ll i$$

i.e. bitwise left shift the most significant bit (MSB) of *i* positions (e.g. CPU1→ $i = 1$ -> 10_B). With the left shift, trailing zero's are filled with zeroes.

5.1.2.3 Alignment Function Implementation

The implementation follows the concept in [Figure 35]. As shown in [Figure 37] the Alignment function first gets the ID of the i-th CPU calling the function, executes the *Align All Cores* for any CPU different from the i and the iterating ones. For example, if CPU2 is the iterating caller, CPU0 is in running mode waiting for the results and CPU1 is an iterating core (a CPU that executes the *Verify*) the *Align All Cores* is called for CPU1 with mask 10_B. However, also CPU2 is iterating hence also CPU2 calls the alignment function and for CPU1 the *Align All Cores* is called. In this case, the mask is 100_B.

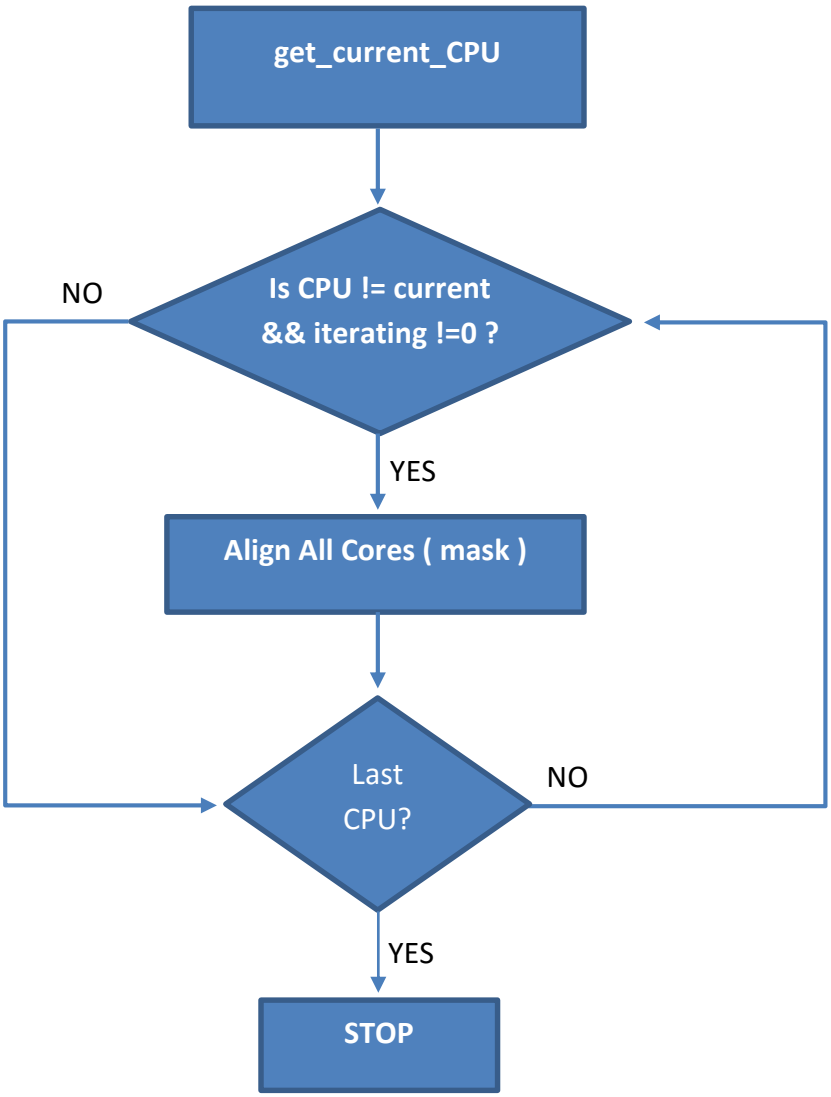


Figure 37: Align All Cores calling flowchart

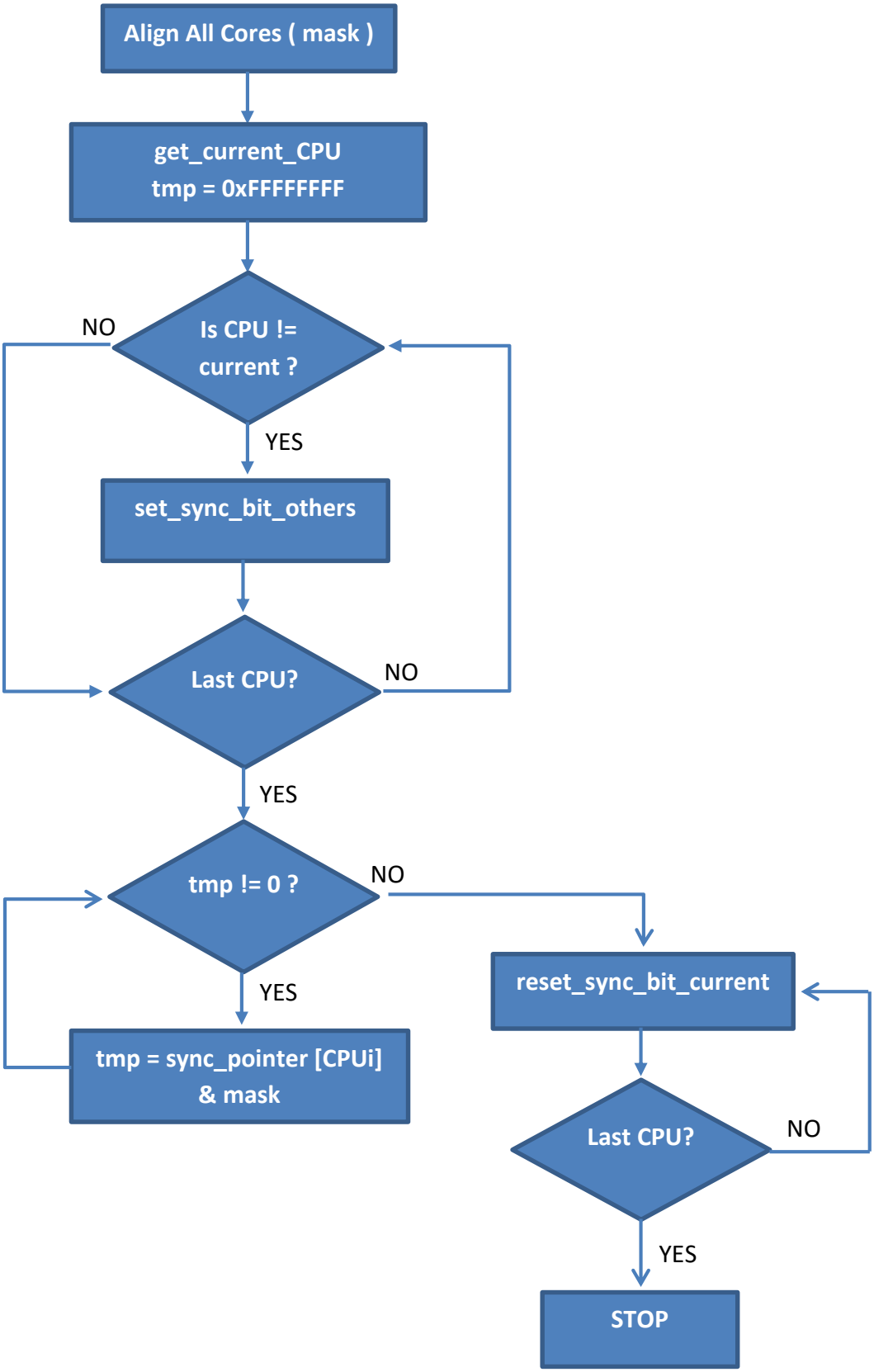


Figure 38: Align All Cores flowchart

When the *Align All Cores* [Figure 38] is called, the first step is to initialize the variable *tmp*. Then the *set_sync_bit_others* function is called. This one sets to zero the bits of the other cores *Sync Pointers*. Returning to the previous example, suppose CPU2 the alignment funtion caller then the third bit from left of the *Sync Pointers* in DSPR0 and DSPR1 will be set to zero (this flag means "I'm aligned!"). After the set operation, the CPU waits with a while cycle until *tmp* becomes zero. This happens when the *Sync Pointer* has the i-th bit set to zero, where CPUi is the other iterating core. If CPU2 is the caller:

$$\begin{array}{l} 111\& \\ 010 = \\ --- \\ 010 = tmp \end{array}$$

until CPU1 (caller in the parallel aligning function) sets to zero the 1. When the CPU exit from this while, reset support function is called and resets to 1 all the last three bits of its *Sync Pointer*.

5.1.2.4 Alignment Function Verification

The *Align All Cores* features have been used in the operating system in the only existing process for now: the *Verify*. Initially, to measure performance in multi-core, the scheduler were disabled during the *Verify* process. This meant that no ISR is executed by the scheduler during *Verify* execution. Under this condition, we measured performances shown in [Figure 49]. A later chapter will study deeper the verification. It is important to keep in mind that this feature, as it is implemented, hides issues.

5.2 Code Cloning

The Infineon operating system was born with the aim of automating the testing, done to characterize and validate the e-Flash memories used in automotive (ATV) control units. The code must be placed in RAM. The TriCore™ microcontroller family, which loads the operating system, contains a master CPU and two slaves CPU. Each of them has its own dedicated RAM [Figure 39]: the Scratchpad RAM. We have already said how is divided the architecture of the latter, for any CPU. At the boot, the FTOS code must be cloned from CPU0 RAM to the other cores RAM. But, not all the operating system code must be cloned: only the variables and the functions executed by CPUs. Flash memory is divided into different sectors and we want to test the sectors with each correspondent core to speedup flash testing.

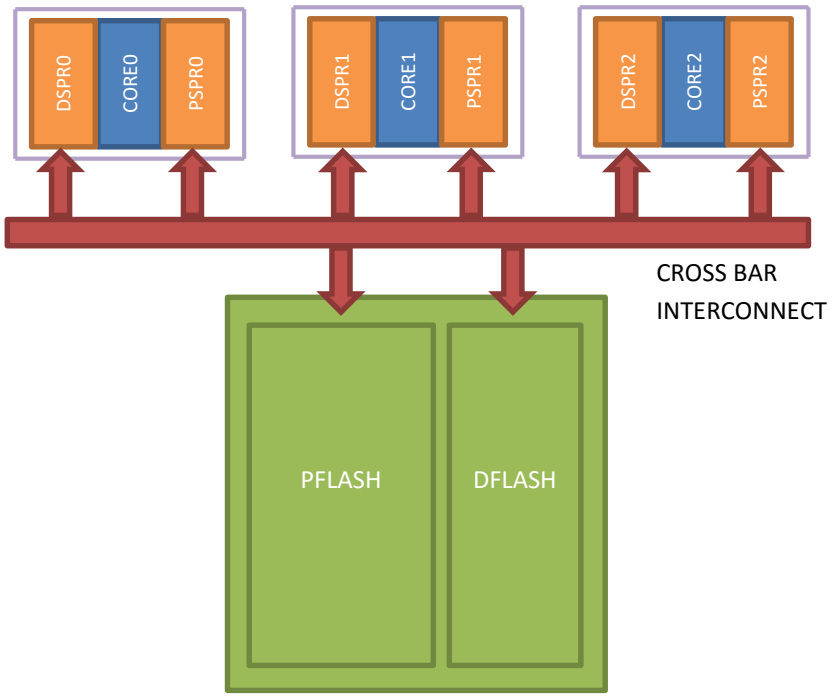


Figure 39: CPUs and microcontroller memories

5.2.1 The C build process

To better define the problem, we must understand the C build process. Every time you write a piece of code, two types of files are used:

1. **Header** files (.h extension): the .h file is the interface to whatever is implemented in the corresponding .c file
2. **Source** files (.c extension): they implement the interface specified in the .h file

If a function is defined in a .c file, and we want to use it from other .c files, a declaration of that function needs to be available in each of those other .c files. That is why you put the declaration in an .h, and #include that in each of them. You could also repeat the declaration in each .c file, but that leads to lots of code duplication and an unmaintainable mess.

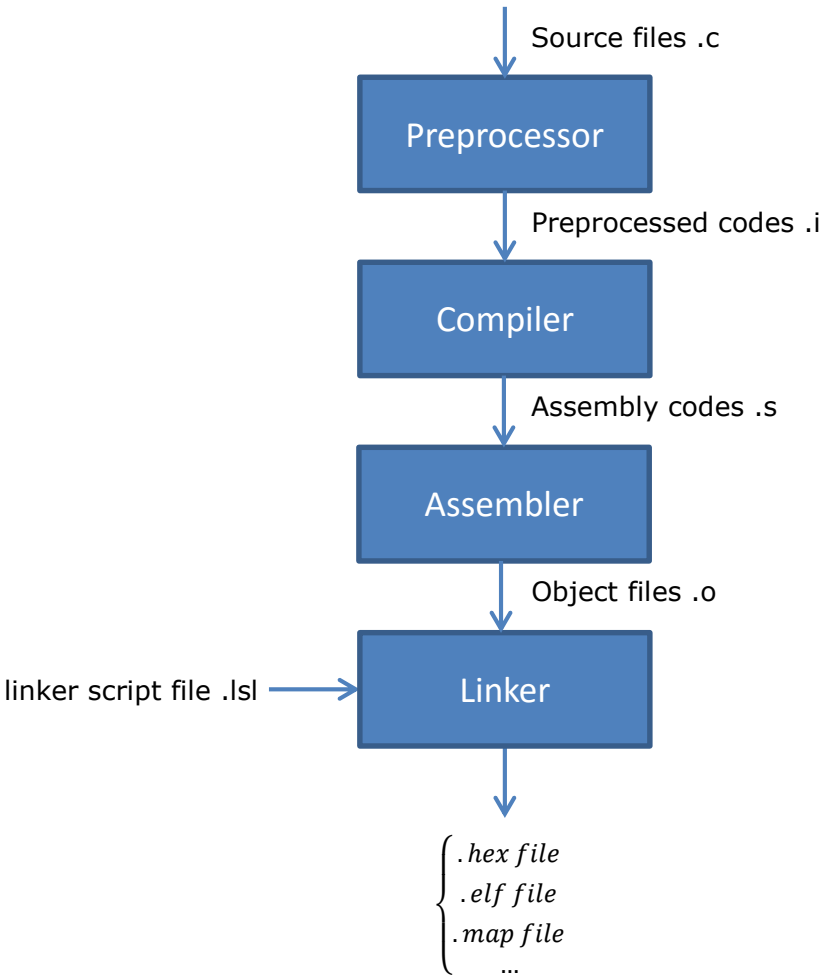


Figure 40: Build process

For definition: a compiler is a computer program that transforms/translate source code (written in some source language) into another computer language called target language [27]. The source language is a high level language (in our case is C) and the target language is a low level language The target language is object code that can be executed by a hardware platform. However, programmers often use the term “compilation” to refer to the build process as a whole. The Altium TASKING compiler is the one used. In [Figure 40] we can see the build process. The C preprocessor transforms C programs with *#directives* to C programs where those directives have been executed and the results are preprocessed code files (.i extension). The main C language features implemented as directives are:

- File inclusion (#include)
- Macros (#define)
- Conditional compilation (#if, #ifdef, #ifndef, etc.)

After the preprocess phase, the codes are compiled giving assembly codes (.s extension) as result. The C compiler is also called *cc*. The assembler translates assembly language to machine code. Assembly is a human readable language but it typically has a one to one relationship with the corresponding machine code. Therefore, an assembler is said to perform isomorphic (one to one mapping) translation. The linker then processes the object files, returned from the assembler, using the notes written in the linker script file to assign absolute memory locations (*locating phase*) to codes and resolve any unresolved references (*linking phase*). The linker script file (.lsl extension) contains a detailed scheme of how the program memory (in our case the RAMs of each CPU) is divided. The linker finally produces a binary executable that runnable from the command interface. The TASKING compiler is able to produce different executable file formats: HEX, ELF, MAP, etc. The MAP file contains a plant of the locations and call functions in which they are located, so that, when we are debugging is possible to understand what function we are performing. The ELF stands for Executable and Linkable Format. It is an executable file format with a dimension larger than the HEX files because it contains also the debug symbols. ELF is flexible and extensible by design, and it is not bound to any particular processor or architecture. Finally, the HEX file format. We must to focus on this last, because the cloning is not possible because we must to fix these files. HEX is a file format of Intel, which conveys binary information in ASCII text form. Intel HEX consists of lines of ASCII text that are separated by line feed or carriage return characters or both. Each text line contains hexadecimal characters that encode multiple binary numbers. The binary numbers may represent data, memory addresses, or instructions depending on their position in the line and the type and length of the line. Below, we can see an example of a line:

```
:10010000214601360121470136007EFE09D2190140
```

1. `10` → 16 in decimal → record length (2 nibble at a time)
2. `0100` → address respect to header
3. `00` → type of record (00 means data)
4. `214601360121470136007EFE09D21901` → record content
5. `40`→checksum

The record is any line starting with ":", finishing with the correspondent checksum and of length defined in the first two nibbles. Any HEX file contains, as first record, the header (codified in the HEX format) that corresponds to the first address of the memory section defined in the LSL file. As last record, there is an EOF (End Of File). Therefore, the second point represents the relative offset to the first record. The record types are six:

00: Data record
01: EOF
02: Extended Segment Address (only for 80x86 processors)
03: Start Segment Address (only for 80x86 processors)
04: Extended Linear Address (allows for 32 bit addressing)
05: Start Linear Address (only for IA-32 architecture)

We only use 00 and 01 record type. The checksum, instead, has the purpose of detecting errors, which may have been introduced during storage. It is computed by summing the two nibbles at time, extracting the least significant two nibbles of the sum and then calculating the two's complement of the LSB (e.g., by inverting its bits and adding one). Let's make an example:

```
:02005000 9A32checksmum
```

First thing, the sum:

$$02 + 00 + 50 + 00 + 9A + 32 = 11E$$

Second thing, extract the least significant two nibbles:

$$11E \rightarrow 1E$$

The third, the two's complement by first converting from hexadecimal to binary:

$$0x1E \rightarrow 0001\ 1110_B$$

Then inverting

$$0001\ 1110_B \rightarrow 1110\ 0001_B$$

Finally summing one

$$1110\ 0001_B + 1_B = 1110\ 0010_B \rightarrow 0xE2$$

Therefore, we obtain:

```
:02005000 9A32E2
```

As we have already said, the ELF file format contains the debug symbols. This is useful for debugging the code that we written but it is useless for the operating system loaded into the microcontroller. HEX files contains only the executable code and are lighter. The TASKING compiler has an option that splits the HEX files for any memory location, but the offsets and the header are not correct. Another important thing, we want a unique HEX file to load into the microcontroller.

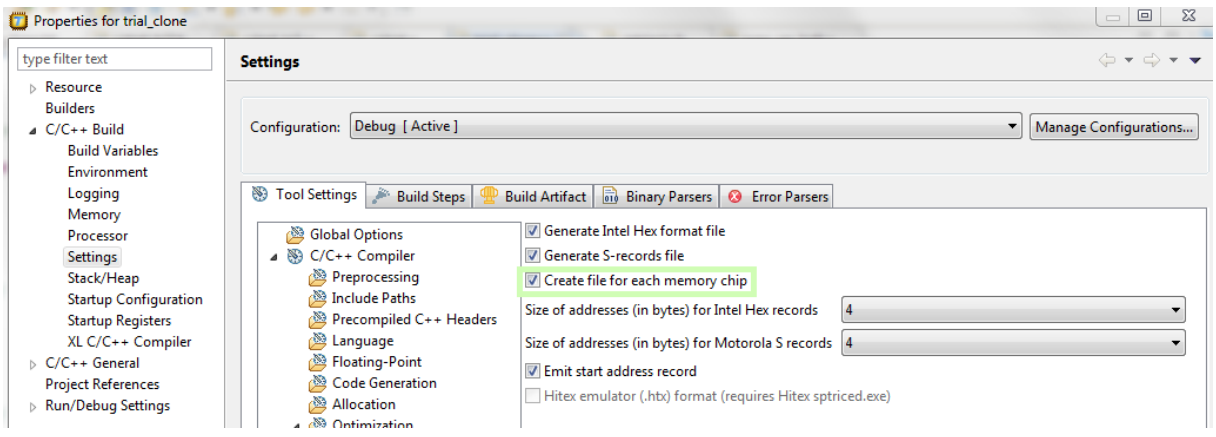


Figure 41: TASKING option for multiple HEX generation

5.2.2 Code Cloning Requirements

1. Code must be loaded in locals SRAMs to speed up execution and leave flash memory empty for self-testing
2. LSL file permit to define the memory sections. The memory areas in the .lsl file must reflect the memory on the core
3. ELF and HEX files must contain the cloned code generated from the linker
4. Load a unique HEX file to the microcontroller that automatically load a copy of the code marked with the command "clone" to any CPU
5. The code in the ELF, must be debuggable with UDE PLS tool

5.2.3 Feasibility Study

In the "TASKING VX-toolset for TriCore User Guide" [28], the reference manual, is written:

The linker recognizes the section names, duplicates clone sections for each binary compatible core and locates core specific code and data in the scratchpad memory of each core, resulting in one absolute object file (ELF) for each binary compatible set of cores.

Thus, it is confirmed that ELF file is correctly cloned over the CPUs. However, the manual does not include any info about HEX file/files. Before we hit the ground running, we decided to analyze the feasibility of certain approaches and then follow the best in relation to the time available to complete the release. These are the approaches:

A. Cloning using the __clone and #pragma keywords + LSL editing

B. Cloning using a *memcpy* command + LSL edit

C. Cloning using a copy table + LSL edit

D. Cloning by splitting the HEX files with TASKING and fix/merge with a Perl script + LSL edit

Approach A)

To testing cloning feasibility, a dummy source file was implemented [Figure 42]. Loading and executing the code below, we tested if it is possible to clone functions and global variables with the __clone() keyword and the #pragma core directive. The *calculateValue*() function simply calculate some dummy values (no matter the resulting value) and writes it in 0x50000000, 0x50000004 and 0x50000008 (some free memory locations).

```
INDIRECT void calculateValue (void);

uint32 __clone counter = 1;

typedef enum
{
    CPU0 = 0
    , CPU1 = 1
    , CPU2 = 2

} FTOS_CPU_ID_t;

int main(void)
{

        if ( MFCR( CORE_ID ) == CPU0 )
        /* Master Core main */
        {
            calculateValue();
//          printf( "Hello world\n" );
        }

        else
        /* Slave Cores main */
        {
            calculateValue();
//          printf( "Hello world slave\n" );
        }

}


#pragma code_core_association clone
//INDIRECT void __clone calculateValue (void)
INDIRECT void calculateValue (void)
{

    while(1)
    {

        counter++;

        uint32 core_id = (uint32)MFCR( CORE_ID );
        uint32* add = (uint32*) 0x50000000;

        *(add + core_id) = 0xF0000000 | core_id | counter ;

    }

    return;
}
#pragma code_core_association restore
```

Figure 42: trial_clone.c

From this code, we expect a result like in [Figure 43]. The first red row of numbers represents the *calculateValue()* results when all three cores are running. We should have the possibility to stop the core that we want, while the others write on own address with the function. The second window shows the counter variable value. It stops, if and only if, all cores are stopped.

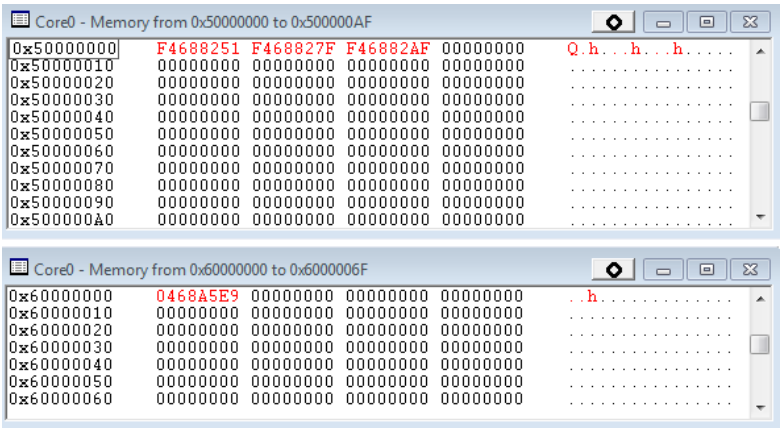


Figure 43: Memory window example in UDE PLS

We tried to explore two principal ways for the approach A:

1) Single .hex file

2) Multiple .hex files

From the single HEX file, we expect, when it is loaded with UDE PLS on TriCore test board, that the code should be cloned over 3 different addresses. But, code is located in one memory location and the bytes shown in [Figure 43] doesn't change. The behavior with __clone() keyword is equal to #pragma directive. No possibility of implementation is the result. With the "*create file for each memory chip*" TASKING tick, there are 3 .hex files, each for any PSPR. Pay attention: before cloning .hex files, we created 3 HEX dump files from the memory windows at any PSPR start address and, copying the first .hex line of any dump, we fix the wrong start addresses of the 3 .hex files generated from TASKING. This solution, when we run the dummy function in UDE, works well, as loading the ELF, only with the function cloned (with either __clone() and #pragma commands). It does not work with variables.

The pros are:

- Works with functions
- __clone() command used on the code works
- Easy debugging
- Multi-core operations are permitted
- Cloning is executed automatically at the startup

However, the cons are:

- Doesn’t works with variables
- We must fix with a script the three header addresses of the 3 .hex files

A clarification: the change of LSL file was applied to all the approaches, because it is the file where the memory areas are the places where code and variables are defined. This file was initially defined and is the same for all approaches.

Approach B)

Even in this case, we use the dummy source file. With memcpy command is possible to copy piece of code. We must define a group in trial_clone.lsl to define the memory location of function (*calculateValue*) that we want to clone.

```
section_layout:tc0:linear
{
    group mygroup(ordered, run_addr=0xc0000400, attributes=r)
{
select ".text.private0.trial_clone.calculateValue";
}

}
```

Figure 44: Group definition

As you can see, the "select" keyword identifies which function is associated with "mygroup" group that is associated with the section of the LSL file called “tc0”. Now that the group is defined in the LSL file, we can work on the dummy source file [Figure 45]. We can call externally the labels that identify the beginning and end of the slice of code that identifies the *calculateValue ()* using. It was defined "size" to identify the difference between beginning and end of the slice. The memcpy, copy the code from _lc_gb_mygroup (b stands for begin) to destination1 and 2 with dimension defined by the size variable. For the test, we imposed that core0 copied the code. This was done by using the #pragma private0 before the *calculateValue*. So, in synthesis, CPU0 allocates the code of *calculateValue()* in destination1 and destination2.

```
        void* size;

        uint32 counter = 1;
        UINT32 data = 0;
        extern __far void * _lc_gb_mygroup;
        extern __far void * _lc_ge_mygroup;


        typedef enum
        {
            CPU0 = 0
            , CPU1 = 1
            , CPU2 = 2

        } FTOS_CPU_ID_t;

        int main(void)
        {
            uint32 size = (uint32)&_lc_ge_mygroup - (uint32)&_lc_gb_mygroup;
            UINT32 destination1 = 0x60100000;
            UINT32 destination2 = 0x50100000;



                if ( MFCR( CORE_ID ) == CPU0 )
                /* Master Core main */
                {
                    memcpy( (PVOID) destination1, &_lc_gb_mygroup, size );
                    memcpy( (PVOID) destination2, &_lc_gb_mygroup, size );
                    data = 1;

#pragma code_core_association private0  /* Specific instance for CPU0 */

INDIRECT void calculateValue (void)
{

    while(1)
    {
        counter++; /* Allocated at 0x50000000 */

        uint32 core_id = (uint32)MFCR( CORE_ID );
        uint32* add = (uint32*) 0x50000010;

        *(add + core_id) = 0xF0000000 | core_id | counter ;

    }

    return;
}
//#pragma code_core_association restore
```

Figure 45: trial_clone using memcpy

With the B approach therefore, HEX file is unique, works well but we must set the program counter address manually for any CPU. This last, is a huge problem because we want an automatic cloning. So, this approach have been discarded immediately. The feasibility study of the first approach was time consuming, especially for analyzing the structure of the HEX files generated by TASKING and to understand what was the root causes of the incorrect file generating. The B approach took less time but the date of the operating system release was fixed.

Approach C)

Just to mention, the third approach is to add some sections to copy table to make the copy of the function done at the startup without implementing it into the code. The *copytable* is a command defined in TASKING compiler. The content is created by the linker and contains the start address and size of all sections that should be initialized by the startup code. For any core, must be defined exactly one copy table in one of the address spaces. Sub-table can be created with table command; each sub-table can be handled separately from the others and from the main table.

```
copytable
(
    align = 4,
    dest = linear,
    table
    {
        symbol = "_lc_ub_table_tc1";
        space = :tc1:linear, :tc1:abs24, :tc1:abs18, :tc1:csa;
    },
    table
    {
        symbol = "_lc_ub_table_tc2";
        space = :tc2:linear, :tc2:abs24, :tc2:abs18, :tc2:csa;
    }
);
```

Figure 46: Example of use for copytable

- ***align***: optionally can define the alignment. This alignment must be equal or larger than the alignment that you specify for the address space itself. If smaller, the alignment for the address space is used.
- ***dest***: destination address space that the code uses for the copy table
- ***symbol***: defines the name for the reference passed to the initialization code that handles the sub-table
- ***space***: a collection of initialization entries generated from a section in an address space redirected to a sub-table by putting the address space name in comma-separated list

The fact of create only one copy table for any CPU is a limit. The feasibility study of this solution has not be done. Nothing can be said about its pros and cons and probably, has the potential of being a good start-up solution. A future work could be to do a feasibility study of the C approach.

5.2.4 Reverse Engineering for Fixing Problem

Approach D)

Reverse engineering is taking apart an object to see how it works in order to duplicate or improve the object[20]. The compiler generates multiple executable files but these files does not load correctly on the cores. The root cause is unknown and we need a feasibility study to understand the root cause and check if this solution is feasible.

The approach used is:

1. ELF file loaded on UDE PLS debugging tool (ELF file correctly clone functions and variables)
2. For each section, was searched the memory window occupied
3. It has been saved a dump file of any window in HEX file format
4. All dump files have been loaded on UDE PLS with the loading criterion: PSPRi/DSPRi on CPUi with i=0,1,2
5. Run

The results of this study are:

- With the HEX files dumped from the ELF file loaded, the code execute properly. So, the dumped HEX files are right formatted
- Comparing any HEX file generated from TASKING with the corresponding dumped .hex file we found:

    1. HEX files have an incorrect header

    2. Any offset starts from 0x0000 instead of the correct offset written in the LSL file

Therefore, the solution is to fix the header and the offset with a Perl script and then load a unique .hex file on the microcontroller.

[20] Software reverse engineering is done to retrieve the source code of a program. Some motivations are the source code was lost, study how the program performs certain operations, to improve the performance of a program, to fix a bug, to identify malicious content in a program such as a virus or to adapt a program written for use with one microprocessor for use with another. Reverse engineering for the purpose of copying or duplicating programs may constitute a copyright violation. In some cases, the licensed use of software specifically prohibits reverse engineering. In our case, simple we used a reverse engineering approach to understand the behavior with and without the problem to determine the root cause and fix it externally.

The “fixing rules” are:

1. Fix the header read from the .lsl file
2. Fix the offset read from the .lsl file
3. Recalculate and fix the checksum code of any .hex file row

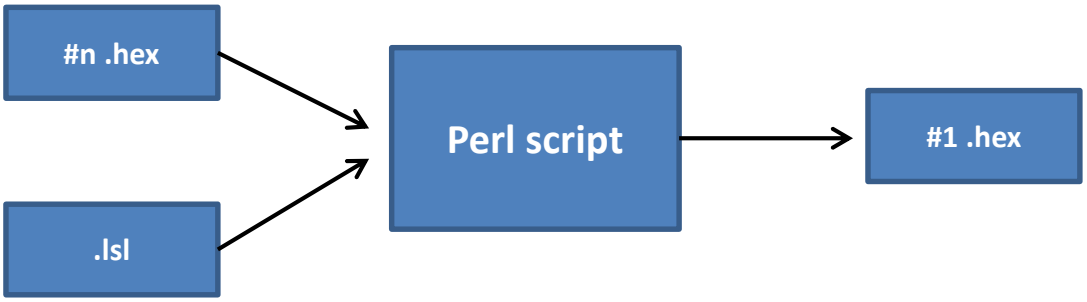


Figure 47: HEX fixing scheme

As can be seen in the figure, with *n* HEX files corrupted and the LSL we obtain a unique HEX file fixed that can be executed on the microcontroller. This script can be used after any new TASKING build to fix the HEX.

Further examination: these fixing rules are respected regardless of the contents of the HEX files? The test performed, to demonstrate that the content does not influence the fixing, is to comment a piece of code in the new FTOS, generate new ELF and HEX files with TASKING and repeat the feasibility study steps.

The results are:

- Fixing rules respected (checked comparing dumped file with new HEX files)
- Operating system initialization sequence appeared in UDE PLS
- Test flow list with dumped .hex files completed and it returns a data log with the same test times that we have with the operating system with all the functions

The Perl script, used for fix / merge the multiple HEX files, is explained in [Appendix A].

This approach was chosen for the constructive simplicity. We have not changed the operating system code. Simply, was written the external script that, after every compilation, is started with a batch file and fixes the HEXs, bringing back a unique HEX. It is a workaround.

A future development could be to continue the approach C study to see if it is feasible and then implement an internal and automatic solution.

5.3 Multi-Core vs Single-Core Measurements

After solving the problem of cloning the code, we collect the performance measurements of the new multi-core operating system comparing those of the old single-core system. These measurements were made with scheduler disabled. The scheduler is the handler of the processes so, multitasking was not possible and tests were never interrupted by interrupt events. The measures are only test time measures. We expect, ideally, that overall test time is less for multi-core than single-core as well as, ideally, any single test performed in multi-core is faster.

Figure 48: JAZZ tool setup

An Infineon standard test flow with JAZZ tool [Paragraph 1.5.3] was launched. This test flow contains a collection of thousands of analog and digital tests. We selected only the tests where the multi-core impacts: the *Verify* tests. The personal computer (where JAZZ software tool is installed) and the JTAG connection form the ATE. The test board connects the microcontroller (DUT) to the JTAG connection [Figure 48]. Any test returns always a PASS or FAIL depending on result of each test. The type of tests performed are: verification of the number of zeros after a memory erase, verification of the number of ones after a memory write of all ones, checkerboard pattern verifications, etc. We cannot provide a detailed list of the tests performed, because are included in the non-disclosure agreement. For this

motivation, also the charts reported below do not report the name of the tests and the values of the measures. However, we can understand how much did affect the multi-core configuration on the timing of tests compared to using a single-core operating system as shown in [Figure 49]. The blue bars marks the test times, for each test, for the operating system in the single-core version. The red bars marks the times measured in multi-core. For each selected test in the graph, there are a blue and a red column to compare. JAZZ measures the test times via software and provides a results log file. Some tests were performed with only one core, others with more than one. The number of cores is selected in JAZZ by changing the LUT [Paragraph 4.3].The criterion by which the cores have been assigned to the corresponding memory bank depends on the tests that must be run. For example, on the two flash banks, a check of zeros is made (after a memory erase) by CPU1 and CPU2 in multi-core. The same test, in single-core, is performed by CPU0 that must scan both benches. This implies that the execution time of the same test is greater in single-core, because CPU0 has much more memory to scan alone.

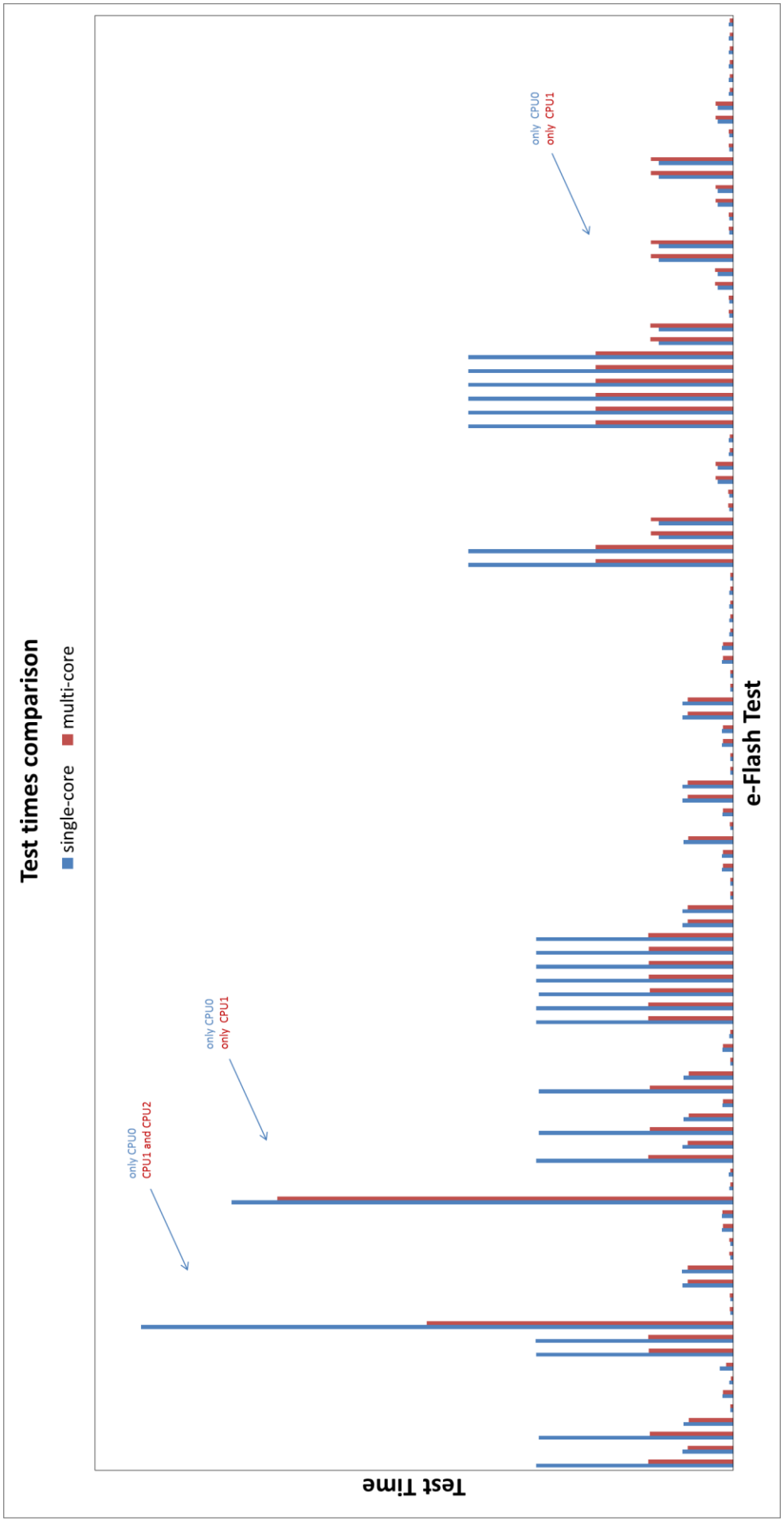


Figure 49: Multi-core vs Single-core FTOS

However, in multi-core as long as the other two cores CPU1 and CPU2 are running, CPU0 is waiting for the results, then merges them and provides a single result log. So where you see "CPU1 and CPU2" in the graph, must be considered that is included the overhead Δt of CPU0 to process the results. If the two cores did not merge the partial results, then the multi core perhaps would be even faster. But, the merge is essential, for CPU1 and 2, to provide the same result as CPU0 that runs alone. The [Figure 50] depicts the situation described. Be careful, Δt is small compared to the tests run by CPU1 and CPU2 but by summing all Δt over all the tests becomes a non-negligible effect.

The first pair of columns from left, indicated by the arrow, in [Figure 49] shows that the test

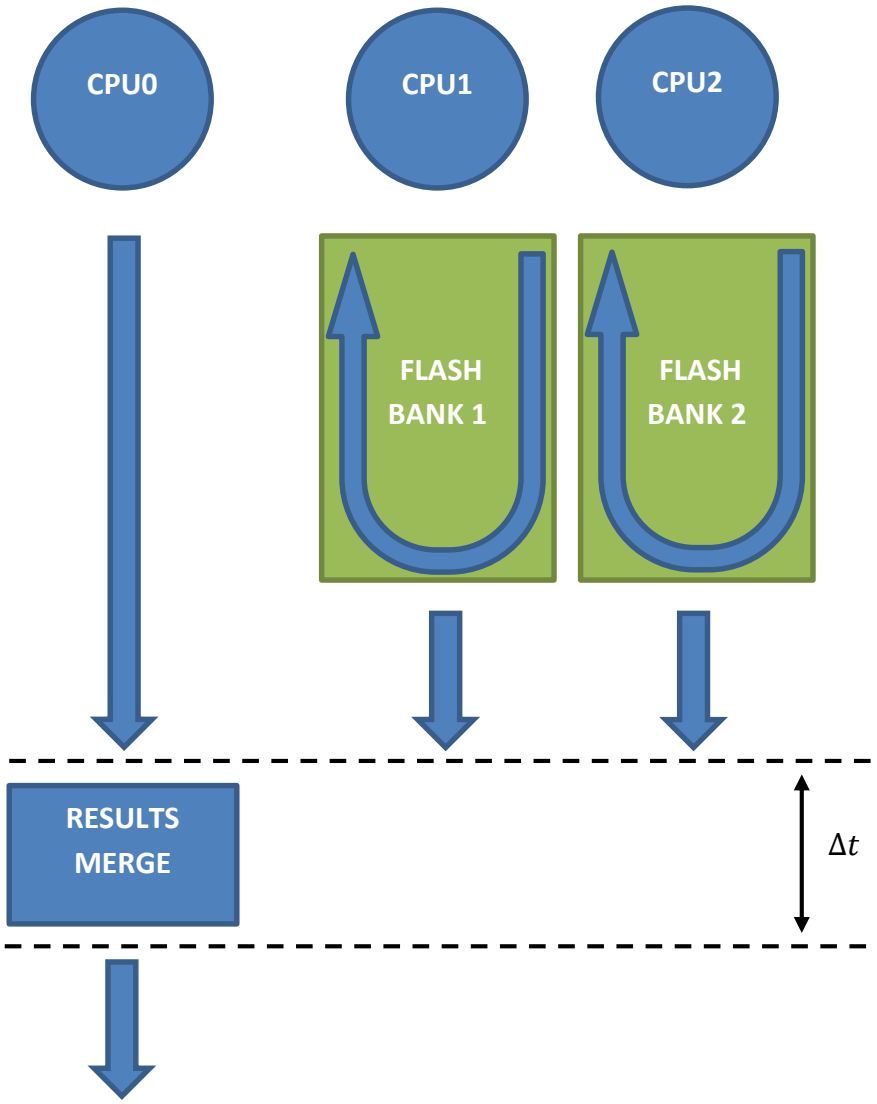


Figure 50: Results merging. CPU0 merges the results of CPU1 and 2

time using a multi-core operating system is approximately halved for the reason just explained. More precisely, the operating system overhead is not only the merging time. Another important contribution to the overhead is the increased complexity of the code for

multi-core. The single-core version has a single CPU that accesses the functions of the operating system. Now the number of CPUs is greater and the architecture is master-slave type, so it is always controlled what cores access to each function. To do this, there are many new if statements and loops in the code that requires more time. An example of what we are saying are the other columns indicated with the arrows. In both cases we perform the tests in single-core (in multi-core, only CPU 1 is selected). If we see the arrow to the left, the multi-core version is slightly faster because CPU1 is a performance core (TriCore TC1.6P). To the right, despite the hardware advantage of CPU1, the single-core operating system runs slightly faster. This is because the test is short and the overhead of multi-core system impacts more. The chart in [Figure 49] shows the impact of any test, but does not give the idea of how much impacts the overall multi-core test time compared to the single-core. By analyzing the collected data, we calculated how much has improved the operating system than the previous version.

	GAIN
Overall test flow	9.61%
Only *Verify*	38.37%

Figure 51: Table of gains

As you can see in the table [Figure 51], we have more impact on the *Verify* tests. Of all tests performed, the impact is about 9%. In [Figure 52] there is the plot of the incremental test time comparison, realized by cumulative summing all the test times. In this way, we can better highlight the gap between multi-core and single-core. Ideally, we expect the curve of multi-core like a translation to the bottom than the single-core curve. In reality, the two curves are slightly different. From this figure, we do not understand how much. We need the next figure to notice better when the differences occurs. The [Figure 53] shows the 100% stacked area chart. The cumulative proportion of each stacked element is always a percentage of the Y axis that is the 100%. For any measure, we have a single-core t_s and a multi-core t_m test time and, in the graph, any blue slices represent $\frac{t_s}{t_s+t_m}$ and any red $\frac{t_m}{t_s+t_m}$ (expressed in %). Pay attention, the graph should be in columns. But, for a better readability, we chose it as a graph of areas. Ideally, we expect the border between the two colored areas is a line parallel to the abscissa. For the first tests this does not happen: it varies from test to test. The table reported a gain in total *Verify* testing time of about 38%. This value corresponds, on average, to the red part of the graph.

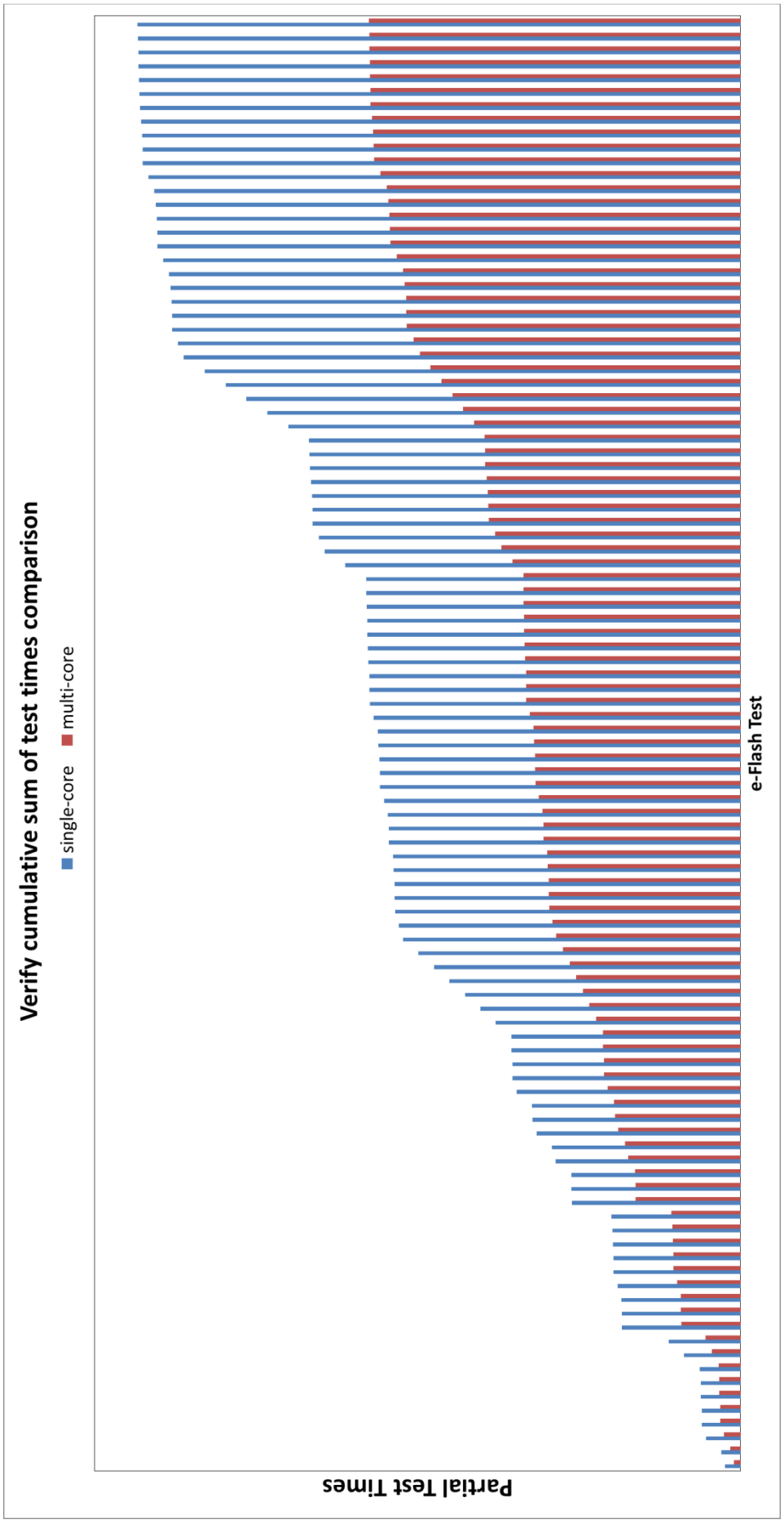


Figure 52: Incremental test times comparison between single-core and multi-core

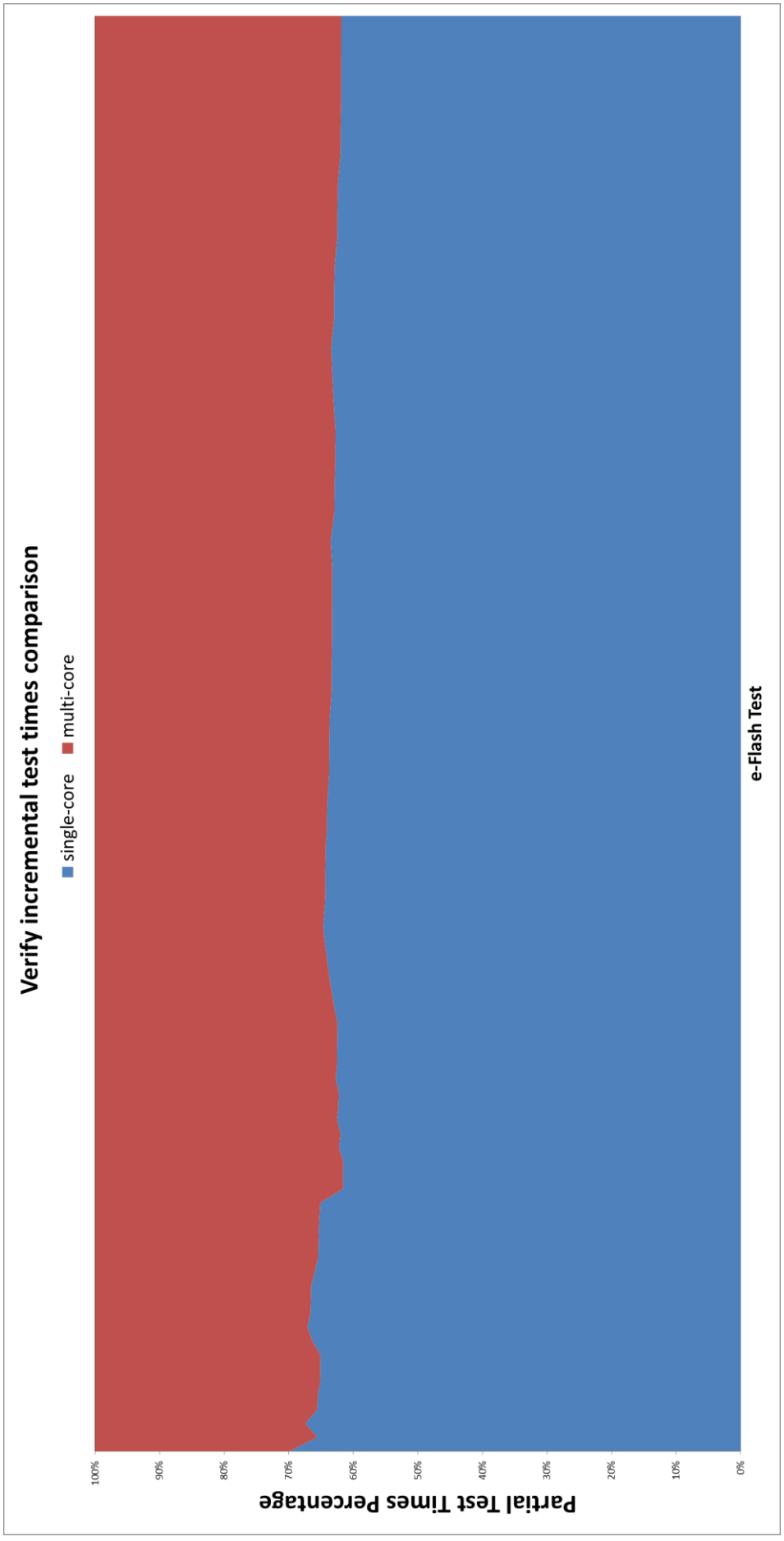


Figure 53: 100% Stacked area chart of the incremental test times comparison

6 Scheduler Characterization

After confirming that the new operating system works well in multi-core mode, let us analyze the multi-task mode. To better understand the scheduler behavior, we must describe the "clock" distribution in the microcontroller. The component that provides the reference clock is the quartz oscillator (*XTAL*) that provides a stable clock signal with frequency $f_{CK} = 20MHz$. Every signal in the microcontroller has a frequency multiple of f_{CK}. The PLL (Phase-Locked Loop) is the circuit that scales the resonance frequency of the oscillator, because the output frequency of the quartz is fixed and low for our application. The microcontroller operating frequency is

$$f_{CPU} = f_{PLL} = K \times f_{ck} = 200MHz$$

where K is the PLL scaling factor. The instruction time is equal to the inverse of the CPU frequency f_{CPU}. The system can select with a multiplexer (MUX) different operating frequencies. In our case, we just know that $f_{CPU} = f_{PLL}$. The frequency divider is set to two by the system registers:

$$f_{STM} = \frac{f_{source}}{N} = \frac{200MHz}{2} = 100MHz$$

The result is the system timer frequency f_{STM} and its inverse is called ΔT_{STM}

$$\Delta T_{STM} = \frac{1}{f_{STM}} = 10ns$$

All these formulas refer to [Figure 54]. The System Timer [Figure 55] content is compared with the Compare Registers to generate interrupts when each System Timer counts a T_{SLICE} time for each CPU. The comparing process is shown in [Figure 56]. The time slice is the time between two interrupts and is a multiple of the ΔT_{STM}. The parameter that scales the ΔT_{STM} to the time slice T_{SLICE} is called TIME_SLICE.

$$T_{SLICE} = TIME\_SLICE \times \Delta T_{STM}$$

We can define that also as

$$TIME\_SLICE = T_{SLICE} \times f_{STM}$$

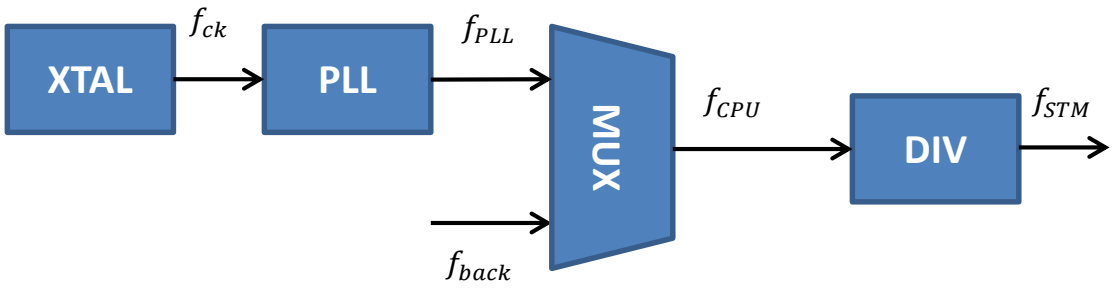


Figure 54: Timing blocks scheme

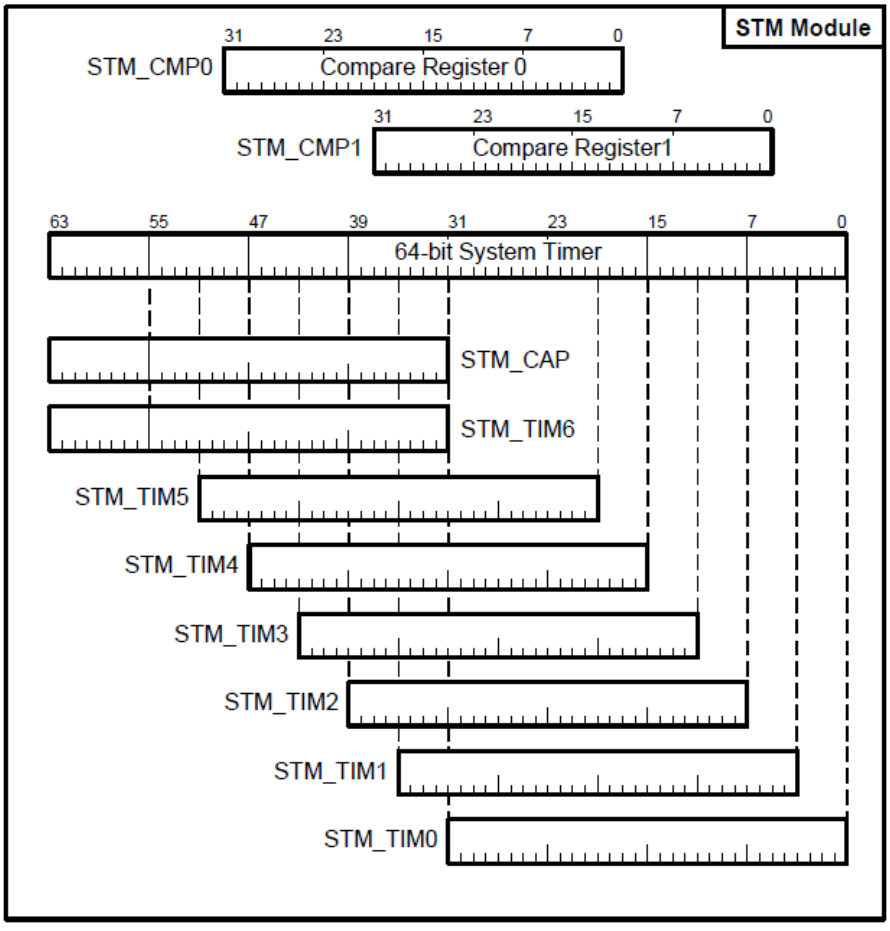
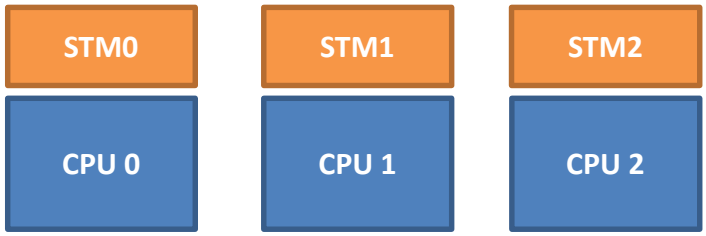


Figure 55: System timer and the two CMP registers (this figure is taken from the TriCore manual)



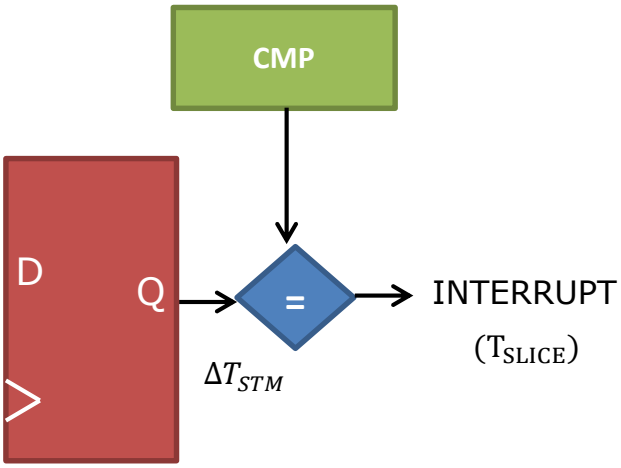


Figure 56: Interrupt generation from the comparing

The scheduler is the software component that represents the core of the multi-task mechanism. This allows the operating system to split in slices the processes and run them in turn, with different levels of priorities. We have already explained how the scheduler works in [Paragraph 4.2]. The key parameter of the round-robin scheduler is the time slice T_{SLICE} (or scheduler step or quantum). This defines the temporal quantum by which a process can perform its task. Each process long more than one time slice continues its task when becomes his turn. We can see an example in [Figure 57]. The *Pi (j,k)* corresponds to the *i-th* process popped from the scheduler processes queue, arriving at *j-th* System Timer tic and with burst duration equal to k tics. The numbers on the time axis indicates the tics and the time slice is equal to 3 tics. Obviously the time slices are not in proportion with the reality; they would be much larger (500-1000 times larger than the tics).

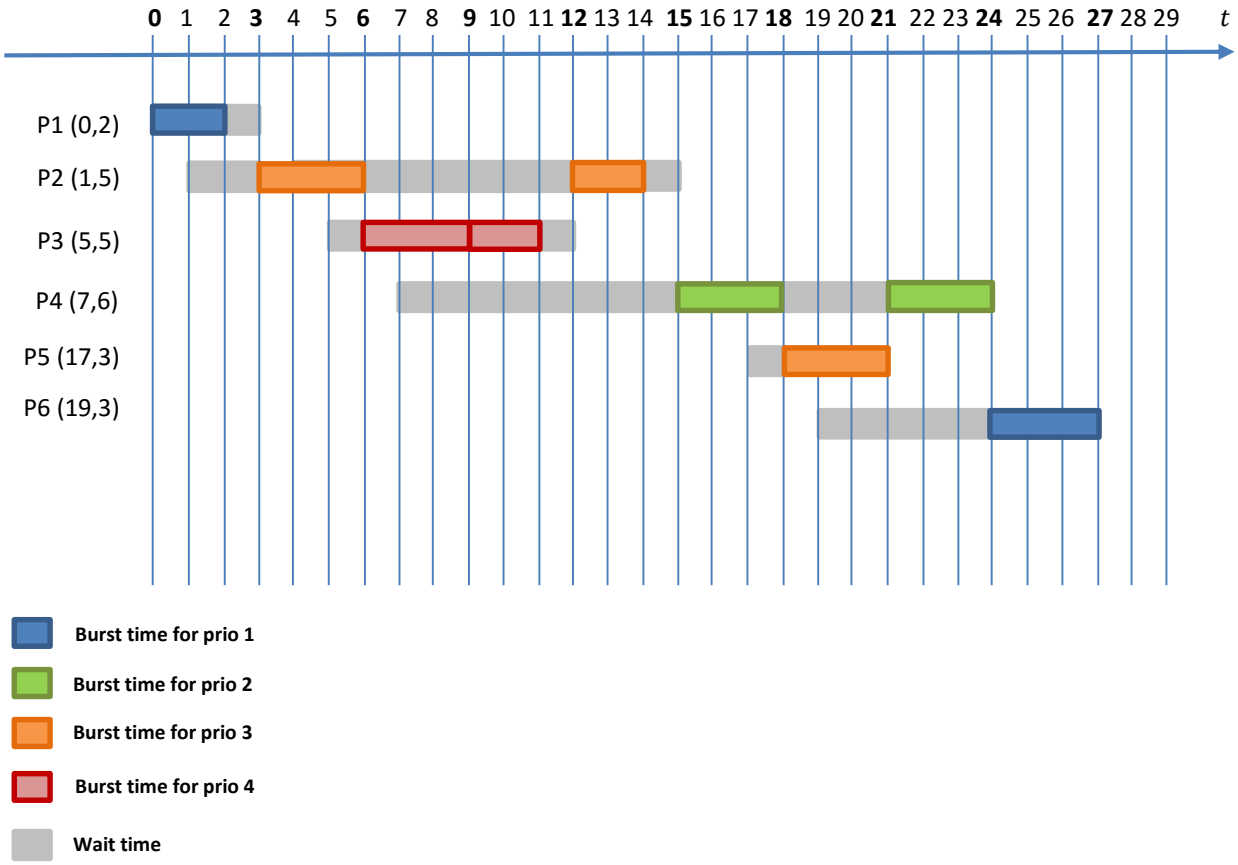


Figure 57: Example of scheduler mechanism

The burst time is the process execution time and can be less than or equal to the time slice T_{SLICE}. The wait time is defined as the time for which the process has to wait to burst or the time it has to wait for the next process. As can we see in [Figure 57], the first process P1 starts at 0 and the burst time is 2 that is less than the time slice (equal to 3). The scheduler wastes 1 tic timer, in this example, before to pop in the processes queue the new process P2. This last, is arrived at the first tic and waits 2 tics before executing. After a time slice, at instant 6, the scheduler checks if there are processes with greater priority: the third process P3 falls in this case because is red. So, the scheduler executes P3 until the next interrupt in 9 (in this case every 3 tics we have an interrupt). Here we have P2 outstanding and P4 waiting for execution, but P3 is the process with the greater priority (prio 4). Therefore, it continues its burst procedure until it finishes and the scheduler waits until the next interrupt in 12. The process with bigger priority is the orange outstanding one and it finishes. Finally the green P4 executes, but it is interrupted by P5 that has a greater priority. P6 is executed at the end because arrives at 19 and waits P5 and P4 that have a bigger prio.

The scheduler characterization studies what happens varying the time slice and checks if the scheduler behaves as we aspect from the implementation. In our study, the operating system contains a single process: the *Verify*. What happens when an interrupt event occurs? Simply the process is interrupted; the scheduler does not find higher or lower priority processes but only the *Verify* and resumes the execution until the end. This is depicted in [Figure 58], the process continues its execution after any interruption and no other processes are present. This representation is not realistic for the proportions and for the duration of the process.

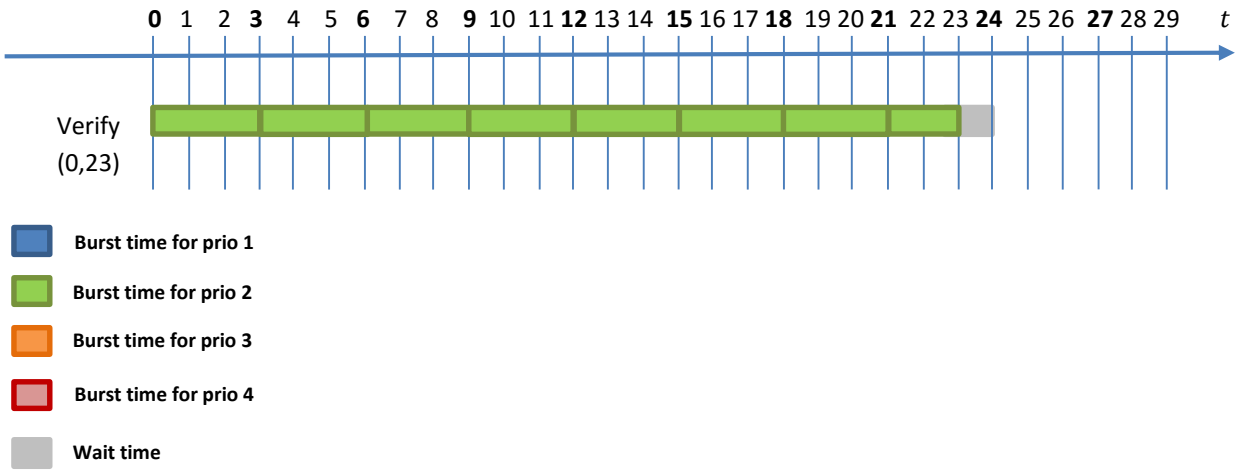


Figure 58: Example of the scheduler mechanism with only one process: the verify

We expect from the time slice change that the shorter the quantum then several times the context switch occurs, because the scheduler has more interrupt events. This implies that there is more overhead on the total test time because the context switch has a duration approximately fixed for each process. Despite we have a single process, the overhead is present and similar because the scheduler pops and pushes the *Verify* in the queue after any interrupt. If we shorten much the time slice, greatly increases the number of pop / push operations in the scheduler and therefore, the *Verify* takes very longer. Of course there is a lower limit and coincides with the value $TIME\_SLICE = 0x1$, that is when the time slice is equal to the system timer ΔT_{STM}. If we increase the value of the time slice, fewer interruptions occur by the interrupt triggered by the comparison between the Compare registers and the System Timer. This means for the scheduler to do fewer pop / push operations and therefore have less overhead on the total test time. There is no upper limit to the value of the $TIME\_SLICE$. The maximum number represented in hexadecimal by a 32 bit variable in the TriCore™, imposes the only upper limit. Having a time slice larger than the duration of the *Verify* ($200ms$ on average) means never interrupt the process and then the $TIME\_SLICE$ loses meaning. Over this upper limit actually the multi-task is disabled. The graph that we expect for the *Verify* test time and of the Total test time at different time slice

must have, approximately, the shape represented in [Figure 59]. We expect the same graph for the total test time and for the individual test time, because, of all tests, the scheduler does the same operations. Then increase or decrease impacts on the operations in the same way. The saturation that we expect is due to the loss of sense of the time slice when it exceed the medium test time. The time slice is a variable defined in the scheduler and sets its granularity.

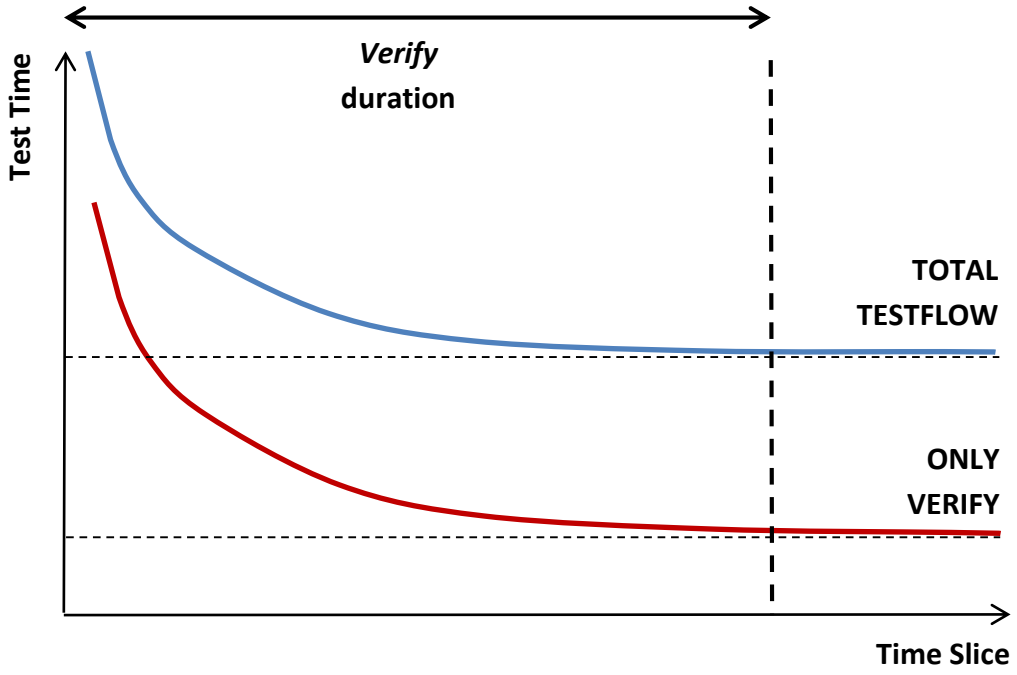


Figure 59: Trends of the test time that we expect varying the time slice

6.1 Measurements Results

The measure procedure is the following:

1. TIME_SLICE modification in the scheduler (varying by increasing the time slice)
2. TASKING compiling (Rebuild)
3. HEX fixing with *tax_generator.pl* . It is the Perl script used for fix/merge the multiple HEX files generated from TASKING compiler [Paragraph 5.2.4].].
4. JAZZ tool setup (update executable files pointers)
5. JAZZ tool flow run
6. *JLTE.pl* run. The JLTE.pl (JAZZ Log To Excel) Perl script is explained in [Appendix B]

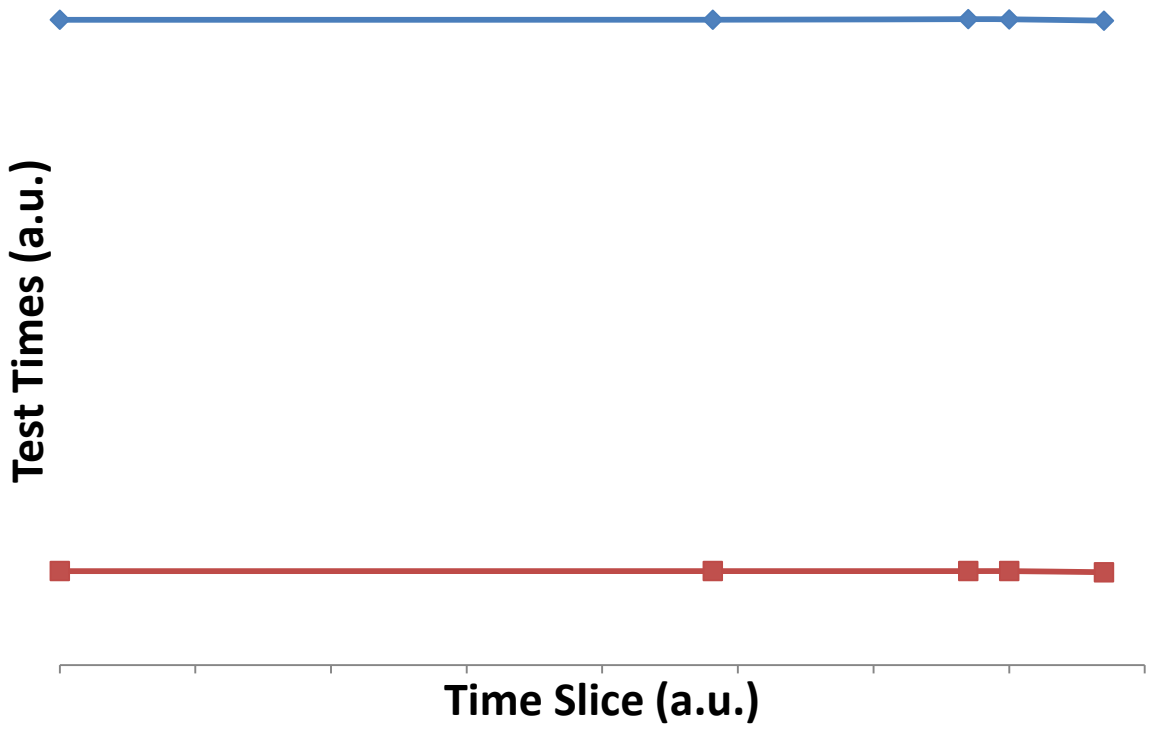


Figure 60: Chart of the times measured varying the time slice

The results collected show that the variation of the time slice leads us to measure the total test times, the *Verify* total times and the average time for each *Verify* as constants. The values are plotted in [Figure 60] in logarithmic scale for the time slice axis. This chart does not reflect what we expected in [Figure 59]. If we process the test times for any time slice, we see that varying the TIME_SLICE implies varying the test times but the test time variation is negligible. So, the test time is not affected by the time slice variation. Something is going wrong. We would try to change our approach and to investigate what happens in the code.

6.2 Code Analysis

We shall try to measure the number of interrupts triggered during the *Verify* tests with different time slices, to characterize the round-robin scheduler behavior. To do this, we insert:

- **Counters** in the Interrupt Service Routine (ISR) in the scheduler (*counter, counter _0, counter _1, counter _2* are the name of the variables)
- **Flags** (*sentinel, sentinel_0, sentinel_1, sentinel_2*) in:
  - *Verify Interface ()*
  - *Verify Extended Sectors ()*
- **Timers** in the *Verify Extended Sectors ()*

In [Figure 61] we can see the *pseudocode* of any source file involved in the interrupts measurements. The operating system code is cloned on each CPU and each core runs its code. Only master core (CPU0) executes the *Verify Interface ()* and that source file initializes the *Verify Interface*, executes the *Verify* and finally terminate the interface returning the errors. Whenever an interrupt occurs in the i-th CPU (to be precise, the compare register has a match with the system timer) the scheduler code is executed. This implies that are first evaluated the if-statements and then is executed the Interrupt Service Routine code. The counters in the scheduler increments itself if the *if-statements* are evaluated as true, i.e. if and only if the sentinels are at 1. For example, if CPU2 has an interrupt event, its scheduler code is called and checks any if-statements. Suppose that `sentinel = 1` because the *Verify* is executing, then `counter` is incremented of one. The `cpu_id` is equal to 2 so the next two if-statements are evaluated as false and core2 arrives to the last if-statement where it increments `counter_2` and execute the ISR code. The variables are saved with an operating system command in a given memory address. So the flags have the task of identifying if the code, after that are set to one, is performing. For any *Verify* executed in the JAZZ flow list the *Verify Interface ()* is called and the counters are reset to 0. We choose an empty memory slice to allocate the results.

```
Scheduler ()

{
cpu_id ← CORE_ID

If (sentinel == 1)
{
save ++counter;
}

If (sentinel_0 == 1 && cpu_id == 0)
{
save ++counter_0;
}

If (sentinel_1 == 1 && cpu_id == 1)
{
save ++counter_1;
}

If (sentinel_2 == 1 && cpu_id == 2)
{
save ++counter_2;
}

ISR CODE (…)

return;
}
```

```
Verify Interface ()

{
counter = 0;
counter_0 = 0;
counter_1 = 0;
counter_2 = 0;

sentinel = 1;

INTERFACE CODE (…)

sentinel = 0;

return;
}
```

```
Verify Extended Sectors ()

{
cpu_id ← CORE_ID

If (cpu_id == 0)
{
sentinel_0 = 1;
timer0 start;
}

If (cpu_id == 1)
{
sentinel_0 = 1;
timer0 start;
}

If (cpu_id == 2)
{
sentinel_0 = 1;
timer0 start;
}

Extended Sector CODE (…)

If (cpu_id == 0)
{
sentinel_0 = 0;
timer0 stop;
save timer0;
}

If (cpu_id == 1)
{
sentinel_1 = 0;
timer1 stop;
save timer1;
}

If (cpu_id == 2)
{
sentinel_2 = 0;
timer2 stop;
save timer2;
}


return;
}
```

Figure 61: Source codes under test

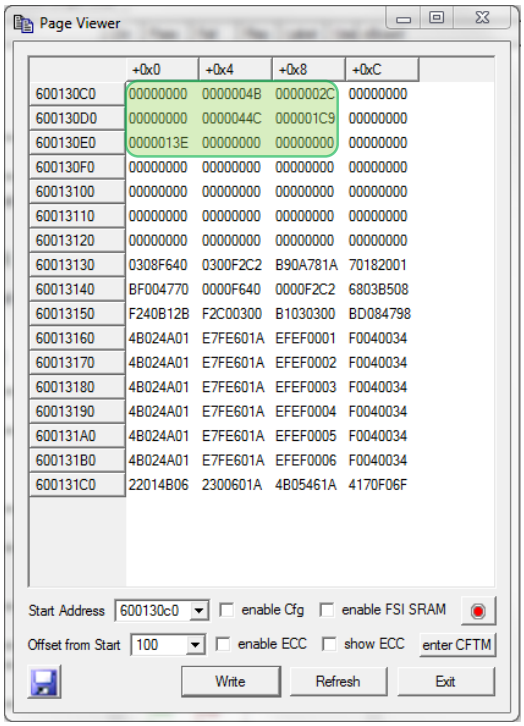


Figure 62: Example of the memory window where the counters and the timers are stored for the tests

In the first line of [Figure 62], we see the number of interrupts of CPU0, CPU1 and CPU2 (from left to right) measured in the *Verify Extended Sectors().* The numbers are in hexadecimal format. The second line returns the *Verify Extended Sectors()* durations for CPU0, CPU1 and CPU2 that we measure with the timers in microseconds resolution. For the last line, should be considered only the first memory address: the number of interrupts measured in the *Verify Interface*. In [Figure 63] we see the results. In the first line we see that core0 is not taking part of the tests. The number of interrupts for core1 and core2 is too small, it seems that the interrupts are disabled. But, this means that we are not executing the code in multi-task mode. The times measured in the second line are the execution time for the *Verify Extended Sectors* of the last test executed and JAZZ returns a test time equal to the one measured with the timers. The number of interrupts measured in the *Verify Interface* matches with the test time.

Finally, the measurements of the first line make no sense and reveal a problem: in the *Verify Extended Sectors* code the interrupts could be disabled.

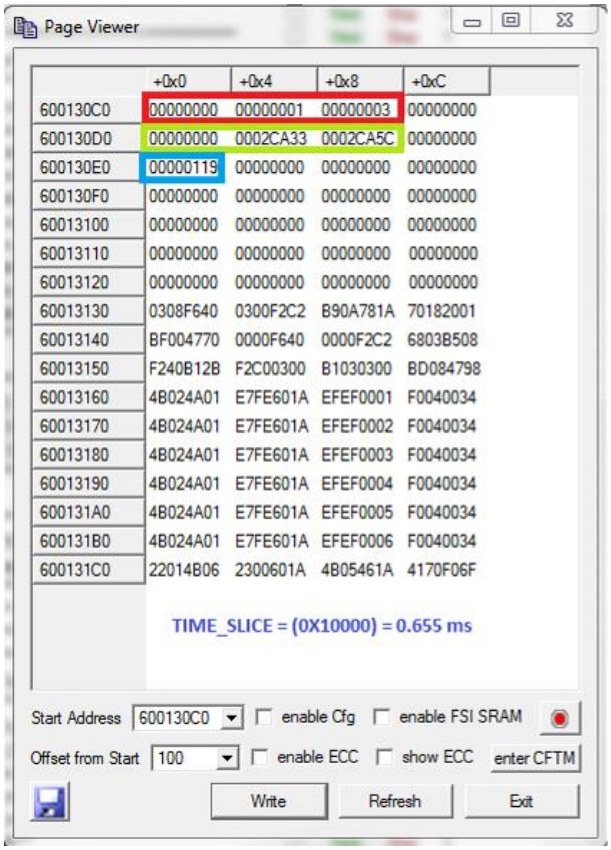


Figure 63: Memory window with highlighted results

So, we analyzed the *Verify Extended Sectors()* code in detail and found that interrupts are disabled at its beginning and enabled at its end. In the multi-core, this disabling has not affected on the verification because every test was performed without being interrupted, so we have performed a pure multi-core test. The tests done varying the time slice for the multi-task verification instead are meaningless, because interrupts are off to the heads of the *Verify Extended Sectors*, then the scheduler executes the code without interruptions. As a result, we enable interrupts to the heads of the function [Figure 64] by simply commenting the disable/enable commands and try to execute. The disable/enable commands were written here, in order to test the operating system in multi-core. This position was chosen to maintain the *Verify* as atomic as possible. An operation is defined atomic when it is not interruptible by any interrupt event during its execution. So the effect of having disabled the interrupts to the heads of the *Verify Extended Sectors* has been to shield it from interruptions.

```
Verify Extended Sectors ()

{
cpu_id ← CORE_ID

Sentinels and timers start(….)

DISABLE_INTERRUPT()

Extended Sector CODE (inside the interrupt disable)

ENABLE_INTERRUPT()

Sentinels and timers stop (…)


return;
}
```

Figure 64: Interrupt disable command to the heads of the verify Extended Sectors

Therefore, one idea is: disable/enable the interrupts, top to down, along the *Verify* function-calling tree. What happens? With the interrupt enabled around the *Verify Extended Sectors*, the flow list in JAZZ stops at the first test. This test is simply a "verify zeros" where the microcontroller use CPU1 and CPU2 to check if, after an erase command over the flash, there are all zeros in the memory. CPU0 in this test waits the results of the other cores for the results merging.

We tried after to disable / enable the interrupts to the heads of the *Verify Sectors ()*. The *Verify Sectors* function is called from the *Verify Extended Sectors ()*. We obtain that the tests execute without problems with the interrupt disabled only on the *Verify Sectors ()*. Obviously, with the disable/enable commented in the code (this means interrupt enabled) the tests blocks at the first test. We tried to disable/enable the interrupt at different levels of the function-calling tree [Figure 65] with different combinations of disable/enable. This solution has brought results meaningless. But, it allowed us to understand that the root of the problem is not in the positioning of the interrupt disabling windows. In some points of the tree [Figure 65], there are some *Align All Cores* functions. This function, as mentioned in [Paragraph 5.1.2], is used to align the cores when they have to set common registers. Since the alignment function has been changed in the concept, it could be the responsible of the issue.

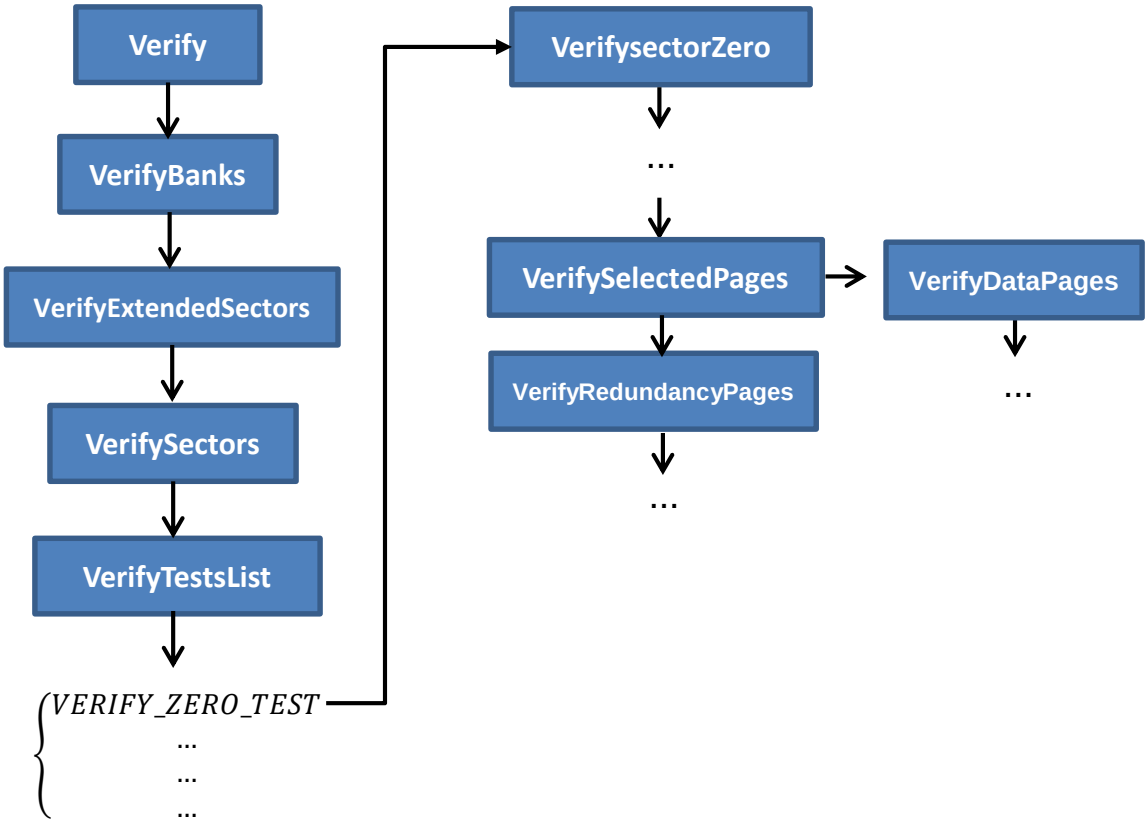


Figure 65: Verify function-calling tree

We try to disable / enable interrupts across the new code that we have written for the *Align All Cores*. We get that the test in JAZZ is still blocked. If we look at the structure of the calling tree we can observe that falls linearly until *VerifyTestsList*. At that point, according to what required by the tests in JAZZ, the test is selected and the call goes down along the tree. At the *VerifySelectedPages* the tree splits. First, the operating system performs the redundancy analysis over the flash memory and then proceeds with the *Verify* on the data. We then tried to disable the redundancy and leave only enabled the *Verify* on the data. The interrupts are disabled / enabled *only* in the *Align All Cores*. In this case, all tests are running in JAZZ. Therefore the *Verify* on the data works properly; the blocks that we see could be caused by something in the redundancy code.

To have the certainty, we have also tried to disable the *VerifyDataPages* and leave only enabled the *VerifyRedundancyPages*: in this case, as we expect, we have that the tests are blocked. All the tests performed until this point, are outlined in the left part of the flowchart in [Figure 66]. After discovering that the redundancy enabled, with disable / enable at the headers of the *Align All Cores* function, implicates having blocks, we wondered if the same behavior is also removing those disable / enable. What we see is that leaving interrupts enabled throughout all the *Verify* code and disabling the redundancy, the tests run without blocks. Therefore, at the beginning when we tested the operating system multi-task by enabling the interrupts, the blocks that we saw was only due to redundancy. So, we

wondered if it was the *Align All Cores* the problem or the redundancy code. Both *VerifyDataPages* that *VerifyRedundancyPages* contain the *Align All Cores* functions. The redundancy of the code has been tested previously; instead the *Align All Cores* has been changed recently. Since the root of the problem is not clear, we decided to proceed with other tests in JAZZ. These tests are listed in the flow chart, on the right side. We focus on the behavioral analysis of the variable *Sync Pointer* of the *Align All Cores* function. This should be 0xFFFFFFFF if the tests are finished without blocking. So, we put ourselves in the case that we have the block and we stop to check the first test of flowlist: the verify zeros. This test keeps CPU0 waiting for the results of the merge as long as CPU1 and CPU2 are running the *Verify*.

When the test is blocked, we see that the *Sync Pointer* of each CPU is different from 0xFFFFFFFF. To CPU0 makes sense because it never makes an *Align All Cores* (not meeting it in the code). Only the two other CPUs modify the *Sync Pointer* CPU0, for this motivation it is always 0xF ... F9.

- *Sync Pointer* [CPU1] = 0xF…FB → 0x1….1|011
- *Sync Pointer* [CPU2] = 0xF…FD → 0x1….1|101

We see that CPU1 sets its bit zero in *Sync Pointer* [CPU2] and CPU2 sets its bit in *Sync Pointer* [CPU1], but it seems that neither has reset its synchronization vector. Using UDE we check the position of the Program Counters of core1 and core2; we see that are blocked in:

- CPU1 → *Align All Cores* (we do not know the position of the *Align All Cores* in the code, for this purpose we must debug step-by-step the code)
- CPU2 → context trap (a generic scheduler trap)

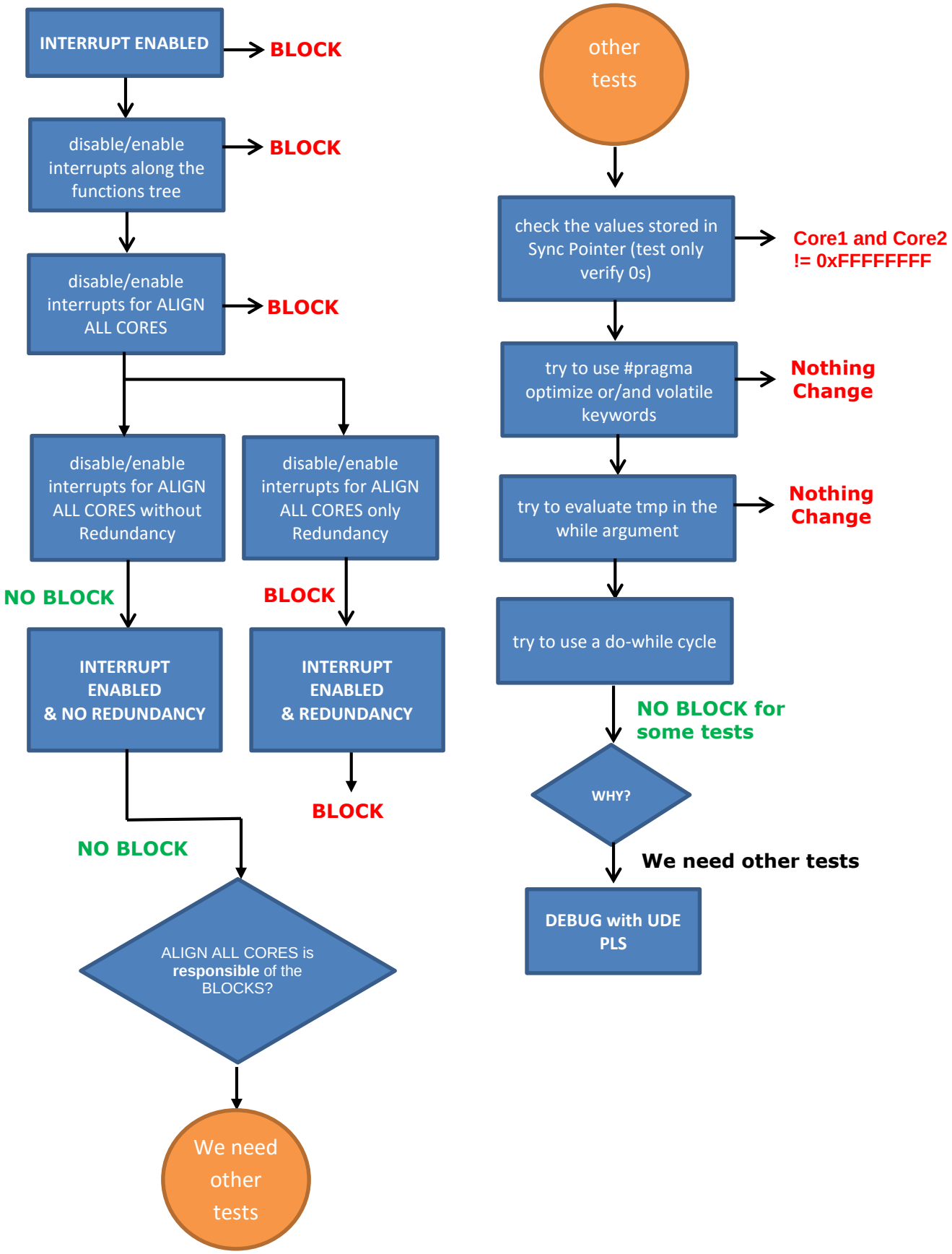


Figure 66: Flowchart of the tests performed

As shown in the flowchart in [Figure 66], we tried other tests before debugging on the board because is more low-level than testing with JAZZ and requires more effort. We tried also to

- Enclose the *Align All Cores* code between the keywords `#pragma optimize` and `#pragma endoptimize`. The pragma optimize must be located outside of a function and is applied to the first function defined after the pragma is detected by the compiler. This avoids the compiler optimizations on the code enclosed by the `#pragma`. This solution does not change the situation because the test blocks itself anyway

- Use the `volatile` keyword. It is used to make sure that we are reading out of the variable is not based on the compiler optimization or an old copy of the variable that the operating system has. This keyword ensures that the variable is fetched from memory on every access. Also this solution does not change the situation

- A combination of these last, but the test blocks itself nevertheless

- Place the *tmp* variable of *Align All Cores* in the while to see if it was a problem related to the fact that while not evaluate the condition in each cycle, after updating the *tmp*. Also this did not bring the test out of the block

- Use a do-while instead of a while loop, the first test (verify zeros) was executed without problems. The second test of the test flow was instead blocked. The third and the fourth ran and the fifth performed was blocked. In short, the do-while allowed some tests to reset the *Sync Pointer* variables of CPUs. This last solution has made us to understand that it was necessary on-chip debugging with UDE.

JAZZ is limited to go in deep then we use UDE to go at low level and check all variable values. The *Verify* code is very complex; we need to reduce the complexity in order to debug step-by-step. We must use the simplest function that emulate the *Redundancy* and all the functions calling from the *Verify Extended Sectors ()* code. The pseudocode below [Figure 67] follows the function-calling tree, in the opposite sense. We used different for-cycles length for any cores involved by using the `cpu_id` that contains the identity number of the CPU calling the function. For example, if CPU2 calls the *Verify Extended Sectors ()* then enters in the first-cycle and runs for three rounds. But, for every lap of the first for-cycle, it makes six other laps. Inside the second for-cycle, there is another for. The code reflects the structure of what happens in the redundancy analysis. As we can see, are performed nested *Align All Cores* functions. The iterator termination permits to kill the iterator status of the cores that call it.

We enable the interrupts for all the following tests, to simulate the blocks and to investigate the causes. With JAZZ, we execute only the verify zeros (core0 wait for merge, core1 and core2 run) and the test (using emulation function and *no iterator termination*) returns:

- Only ALIGN 1 enabled → EXECUTE
- Only ALIGN 2 enabled → STOP
- Only ALIGN 3 enabled → STOP
- Only ALIGN 2 and 3 enabled → STOP
- Only ALIGN 1 and 2 enabled → STOP

The iterators terminate functions are only in *Verify Extended Sectors* and *Verify Sectors* and, for these tests, we do not use it. The cores stop when we have nested *Align All Cores*. The fact that we have not performed the test with the iterator termination enabled, causes core1 to finish before core2 and to remain iterant. Thus, the *Align All Cores* mask does not exclude core1 from the set of core2 *Sync Pointer*, but it cannot do the set because it has finished. Core2 then get blocked in a previous *Align All Cores*. This highlights the fact that a misalignment, in the *Align All Cores* calling, due to different number of cycles could be the root cause.

```
Verify Extended Sectors ()
{
cpu_id ← CORE_ID
for (i = 0; i < 0x3; i++)
      {
      for (j = 0; j < 0x2*(cpu_id + 1); j++)
            {
                        save (address1 + 4*cpu_id) = ++cont_1;

                        Align All Cores;  ← ALIGN 2

                        for (k = 0; k < 0x2*(cpu_id + 1); k++)
                              {
                              save (address2 + 4*cpu_id) = ++cont_2;
                              }

                        Align All Cores; ← ALIGN 3

            }
            Align All Cores;

            save (address3 + 4*cpu_id) = ++cont_3;
      }

Align All Cores; ← ALIGN 1


iterator termination;

return;
}
```

Figure 67: Extended Sectors and Redundancy emulation function pseudocode

With the iterator termination enabled, the test execute in all cases hiding this fact. With UDE PLS, we ran the verify zeros step-by-step, using the emulation function with only ALIGN 2 enabled. The *iterator termination was enabled*. The results agree with JAZZ; the executed test returns:

- *Sync Pointer [CPU0]* = 0xF…F9 → 0x1….1|001
- *Sync Pointer [CPU1]* = 0xF…FF → 0x1….1|111
- *Sync Pointer [CPU2]* = 0xF…FF → 0x1….1|111

After this confirmation, we performed a step-by-step debugging with UDE PLS running the real *Verify Extended Sectors ()*. We ran the test with UDE PLS and, breaking the cores we saw that core2 blocks itself in the *Align All Cores* in the *Verify Cluster Redundancy* (called from the *Verify Redundancy Pages*). So, we stopped all the cores and re-run the verify zeros test. But, this time we put a breakpoint after the first *Align All Cores* in *Verify Cluster Redundancy*. The results are:

CPU1	**HALTED**
CPU2	**RUNNING**

CPU1 is halted because has reached the breakpoint (cloned at the same code point due to the code cloning over the CPUs). CPU2, forcing a break, was in a infinite loop in the *Align All Cores*. This confirms that there is a misalignment on the *Align All Cores* calling. We re-start CPU2, we set it as running after setting the local variable `tmp = 0` (because it was different from zero due to the fact that CPU1 didn't set its bit in the *Sync Pointer*) and we set the Program Counter to the return statement in the *Align All Cores*. The code gets out in a function that writes in the Special Function Registers during the redundancy analysis.

This analysis confirms again that the two cores do not line up in the same level in the code. At least one of the cores, during the redundancy, can exit from an *Align All Cores* to the next level and leaves trapped in the preceding *Align All Cores* the others. In our case the process is unique, because the scheduler has only to schedule the *Verify*, but every core executes its *Verify* that accesses at the variables shared between the CPUs. Each iterating core waits in the *Align All Cores* that the other iterating cores sets the zeroes in its *Sync Pointer*. The situation of deadlock occurs if these four conditions hold true all simultaneously:

1. Mutual exclusion – The i-nth bit, in any *Sync Pointer*, can only be used by a core at a time (the i-nth core)
2. Hold and wait – any CPU waits its *Sync Pointer* to be set
3. No preemption – no CPU has priority over the other
4. Circular wait – all the iterating are waiting for its *tmp* variable to be set, which is set by the other cores

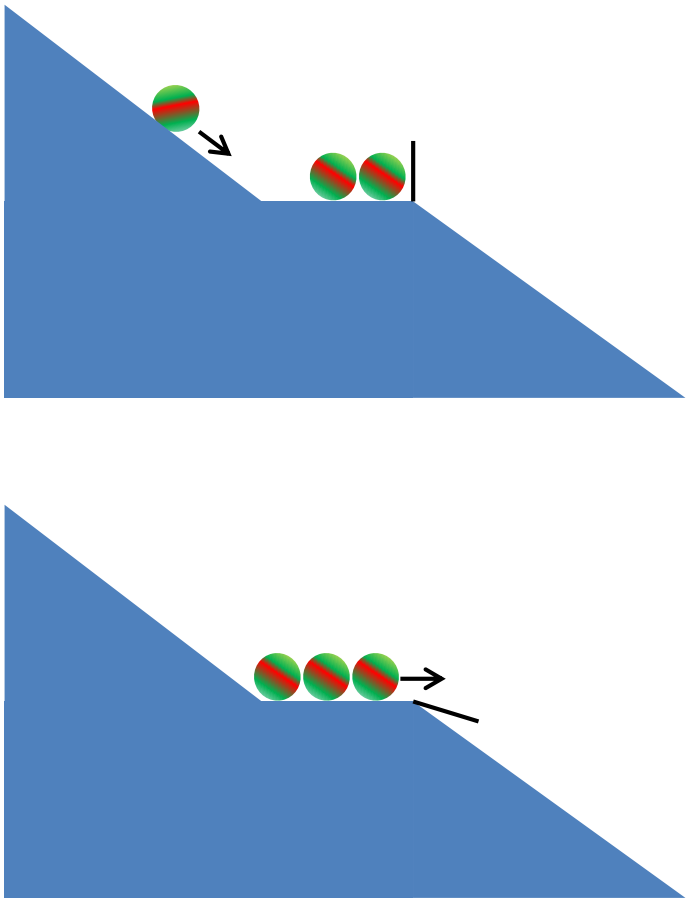


Figure 68: Marble circuit analogy

Al these are true. The deadlock ending requires external intervention. In fact, we exit from the block if and only if we set externally the tmp to zero. In a deadlock, no process can make progress. In our case if a CPU goes to a different level and does not edit the *Sync Pointer* of the others, the other cores make no progress.

6.3 A New Concept for the Aligning Function

The new concept that we propose requires a little effort. We can maintain the same *Align All Cores* of [Paragraph 5.1.2], except for a small modification, and we have to add a new function that keeps track of how many and which core are accessing to it. The problem, as we saw in the code analysis, is related to the CPU misalignment when are running the tests. They call a different number of *Align All Cores* in the code, at the same level. When a CPU runs out before the other, jumps to the next level and sets shared registers instead of waiting the others. We could solve the problem by using the “barriers” concept explained in [29]. E-Flash memory is divided into several parts: banks, extended sectors, sectors, word-line clusters, word-lines, pages, bytes and finally bits. Let us consider, for simplicity, the case with two CPUs. Suppose we are in a sector which core1 has fewer sections to control than core2. Then core1 also come into fewer *Align All Cores* functions. When core1 ends in it

level, goes to the next-level *Align All Cores* function without stopping. We therefore need a function that imposes to core1 to wait until the other CPU does not reach the same function (finishing his own tests). We can consider the CPUs like marbles, the function-calling tree as a circuit, the code as the drops and finally the barriers as the "barrier" function to be placed in the *Verify* code. We can see the analogy in [Figure 68]. The black barriers open if and only if all the cores are at the same barrier. These barriers must be placed at the end of any level (i.e. Word Lines, Sectors, etc.) to avoid that the cores goes freewheeling to the next *Align All Cores*. Ideally, this barrier must open itself if the number of cores calling the “barrier” function is equal to the number of iterating cores. However, this could returns other blocks because if a core arrived at the barrier is still iterating but is not setting its bit in the other’s *Sync Pointer*. So, the barrier must keep track on which cores are arrived. The marbles on the plane are the “waiting cores” that have reached the “barrier” function, are still iterating cores but must be excluded from editing the other’s *Sync Pointer* (for this we must modify one simple thing in the alignment function). The marble descending is the core that is still iterating and executing its test (it has others *Align All Cores*).

Now that we have a clear idea of the problem and a simple model to understand the misalignment, we can list the requirements:

1. **All cores must waits at the same level**
2. **Waiting cores must wait the other iterating cores**
3. There might be a unique and shared among all available CPU(s) (e.g. 3 CPUs) alignment function
4. Function code must be duplicated for each CPU
5. All current alignment functions must be replaced with the newer one

The first two requirements are new; the others are the same of [Paragraph 5.1.2]. The barriers allow us to impose the waiting at the same level and some conditional modifications permits us to reach the requirement number 2.

To have the barriers, we must define a new function that we call “Barrier”. This function simply use a new variable called “Stop Pointer” that contains the flags of the cores that has stopped, entering in the Barrier. We must have a *Stop Pointer* variable for any core. So, the number of *Stop Pointer* is equal to the number of *Sync Pointer*, which is equal to the number of cores. Finally, now, we have $2n$ status variable where n is the number of cores. The *Stop Pointer* is a 32-bit variable, initially equal to 0xFFFFFFFF and when the i-nth core reach the Barrier must set a 0 only on the i-nth position in the other cores *Stop Pointer*. When this is done, the i-nth core must wait by polling its *Stop Pointer* bits until this variable, properly masked, has only zeros: this means that all the iterating cores has reached the Barrier and

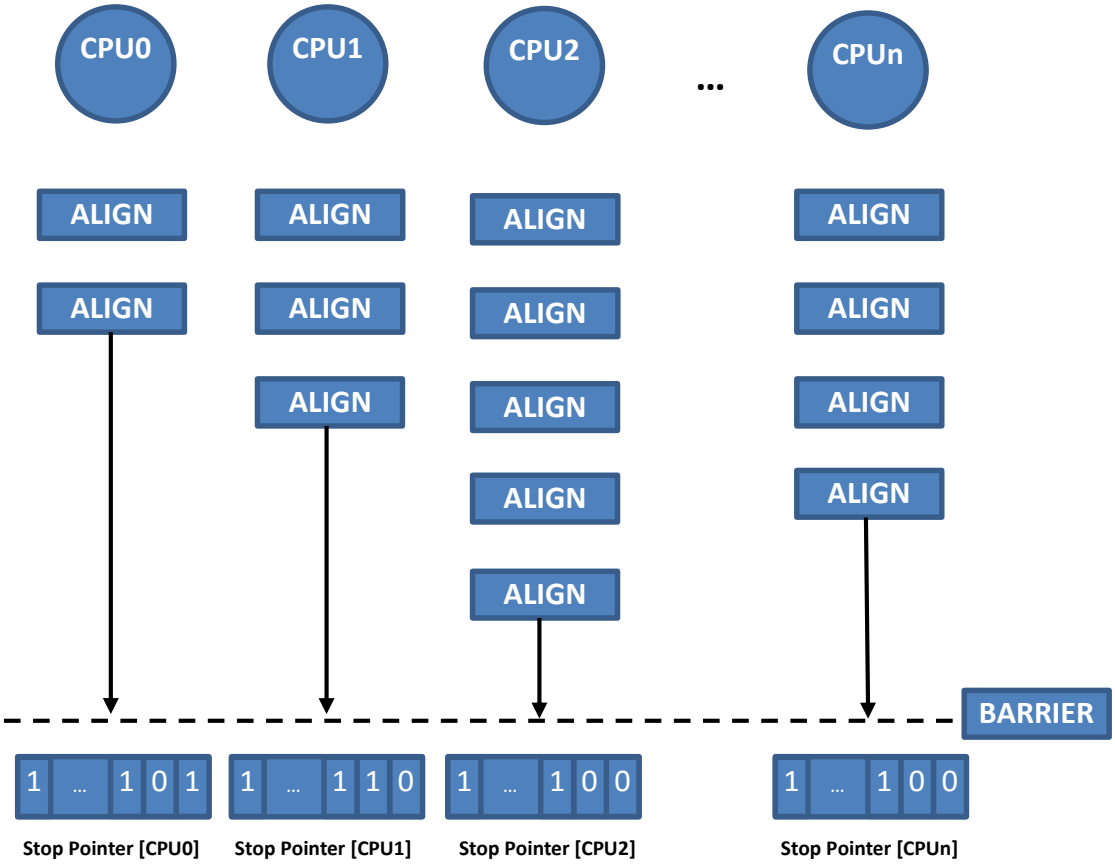


Figure 69: Example of Barrier where only CPU0 and CPU1 are arrived

the i-th core can start testing the next level. An example helps to clarify these last phrases. In [Figure 69], we see that the CPUs hit a different number of *Align All Cores* (ALIGN in the figure). CPU0 is the first ending the ALIGNs and calls the barrier. It sets the zero in position 0 in the *Stop Pointer* of the other cores, and expects its *Stop Pointer* has only zeros in the iterating core positions (except himself). When all the CPUs have called the Barrier this condition will be fulfilled and the last iterating core will sets its bit in *Stop Pointer* [CPU0]. CPU 1 finishes second, and sets the bit 0 in the position 1 of each *Stop Pointer* (except his own) and waits by polling on its *Stop Pointer*. With this mechanism, the CPUs are polling always on its own variable. The set is on the other's bits and no one can set the bits except for the corresponding CPU. This mechanism makes the code conflict free and is easy scalable with more CPUs.

6.4 New Aligning Function Implementation

Below there is a code implementation in flowchart [Figure 71]. We must define a *Barrier* function, which must set, wait and reset. The structure is therefore the same as the *Align All Core*; only the function name and the variables name changes. The *mask* is the same because use the iterating cores. The *Align All Cores* function is similar that we have

implemented in [Paragraph 5.1.2]. Only we have to change the condition in the first decision block (from the top) in [Figure 37]. In [Figure 72] we see the change highlighted.This condition takes into the account also if the *Stop Pointer* has been modified.

1. If the “current” bit in *Stop Pointer [cpu_id]* is 1, the “current” CPU is still executing the code

2. When the “current” CPU reaches the *Barrier*, sets 0 also in *Stop Pointer [cpu_id]* in position “current”

3. When point 2 occurs the if-statement skips to the evaluation “Last CPU?” in the flowchart in [Figure 72]

This new facility allows to exclude from *Align All Cores* the CPUs that have reached the *Barrier* and then reduce the size of the local variable *tmp* only to cores that are still running the tests and that have not reached the *Barrier*. The *Align All Cores* is the same, how can be seen in [Figure 73].

In conclusion, with a small effort, we can modify the alignment function of the cores and probably eliminate the blocks. This must be verified by running the tests in JAZZ, after implementing the code shown in the flowchart. If we not changed the condition in the *Align All Cores*-calling function, we would simply have two *Align All Cores* below and cores that would reach the *Barrier*, not resize the *Sync Pointer*. With this extra condition, we avoid the blocks.

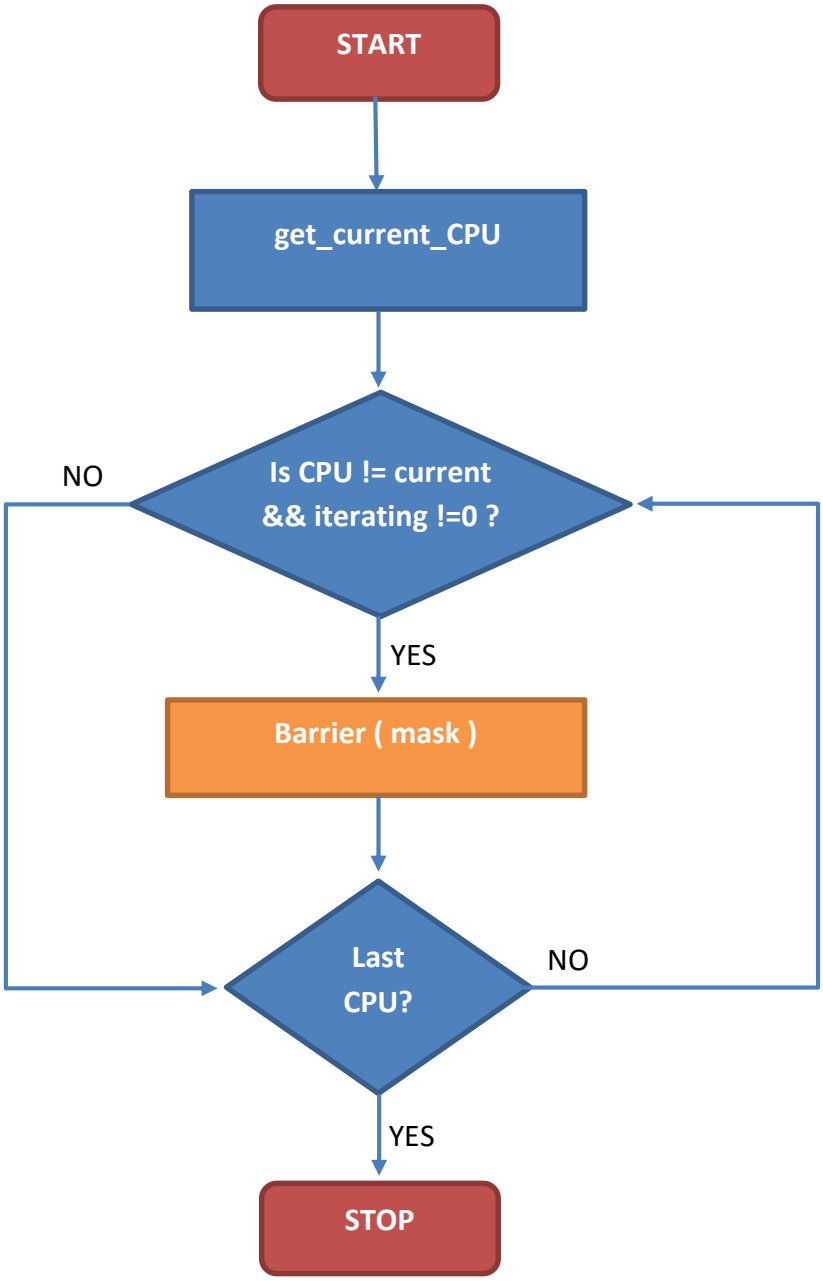


Figure 70: Barrier-calling flowchart

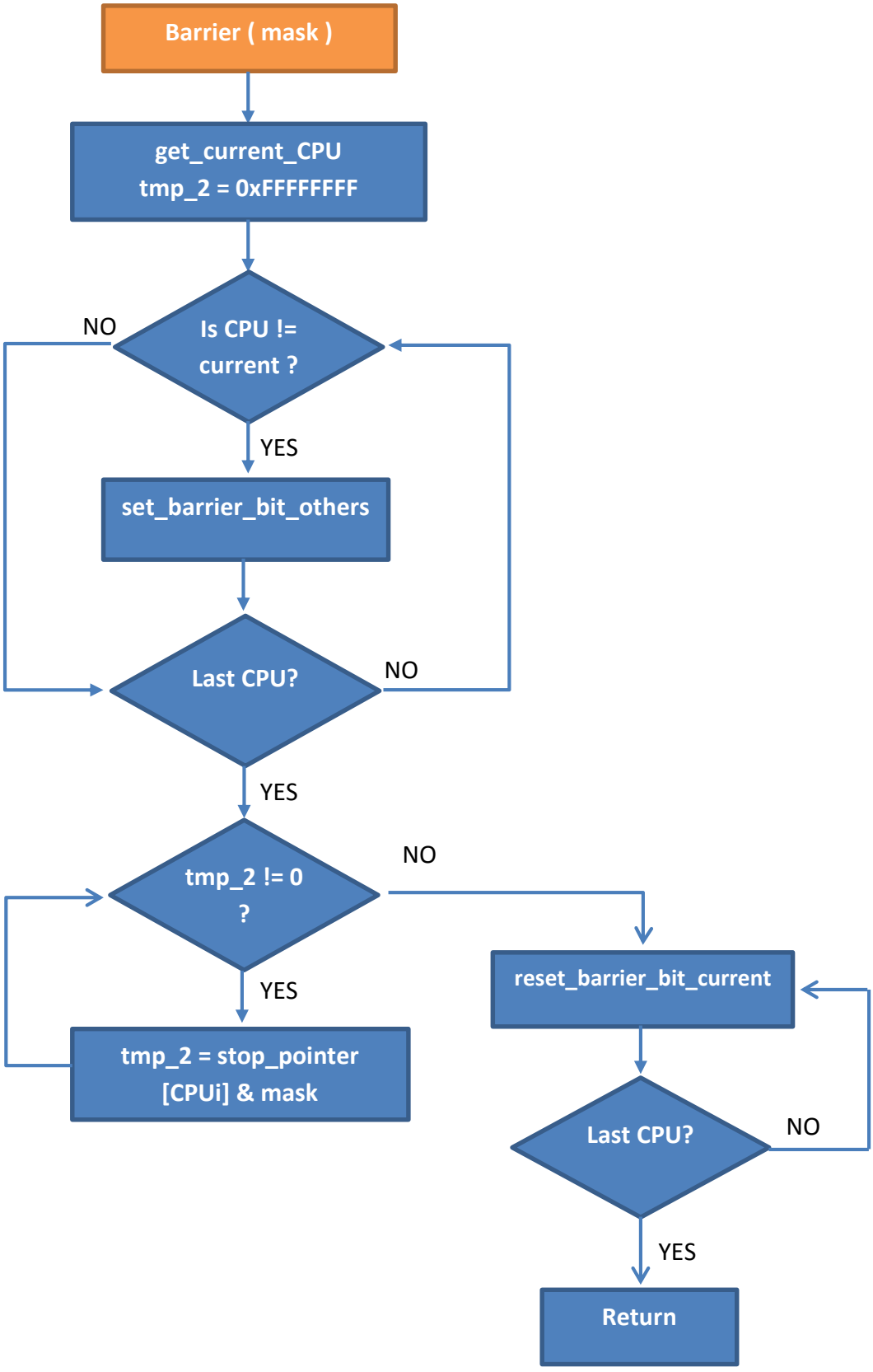


Figure 71: Barrier flowchart

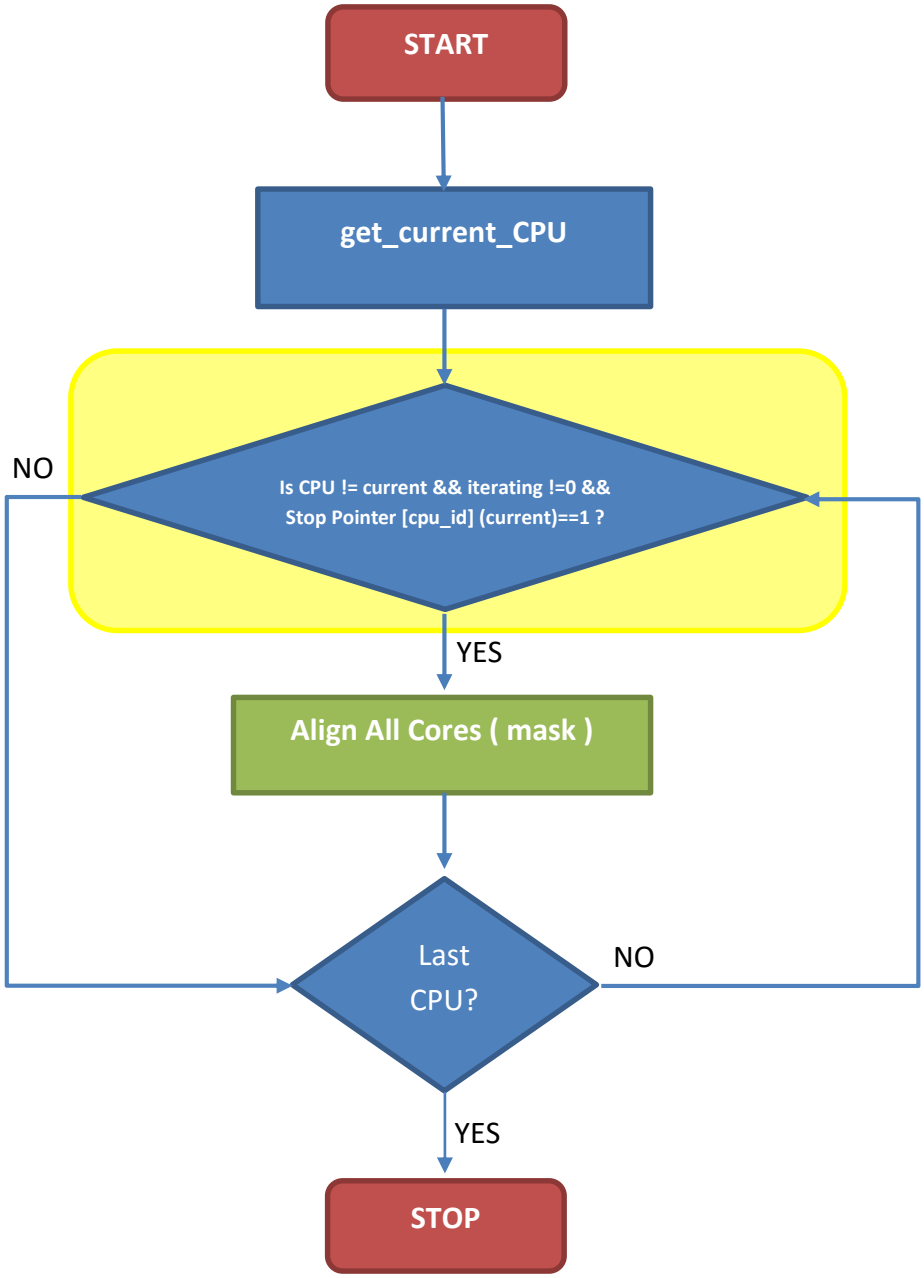


Figure 72: New Align All Cores-calling flowchart

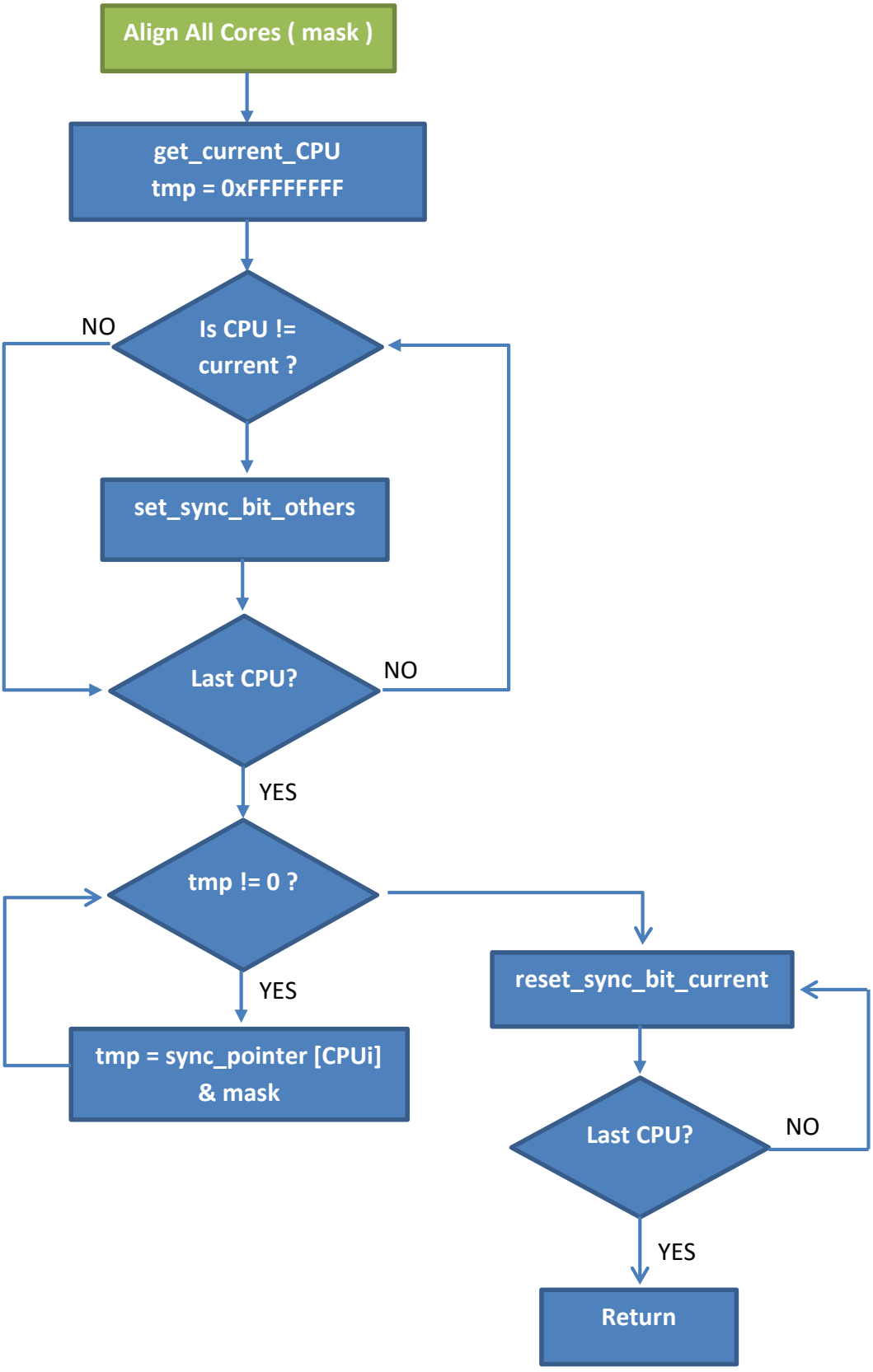


Figure 73: Align All Cores flowchart

7 Conclusions

The presented work is the result of a six months internship at the Infineon Technologies Research and Development Center located in Padua. The thesis goals were to contribute to the operating system Beta release and verify the multi-core multi-task FTOS. The Beta release composed by

- New mutex mechanism definition and implementation
- New alignment function definition and implementation
- Code cloning over different CPUs

has covered about half of the work, approximately 3 months. Last 3 months were spent to verify the multi-core and the multi-task. This last, took approximately all the 3 months.

Achieved results are:

- FTOS Beta release contribution
- Multi-core verification (the test time is approximately halved[21] as we expected)
- Found a bug in the alignment function, avoiding microcontroller market release with incorrect testing
- New alignment function to fix the misalignment issue proposal

The thesis work has required several expertises:

- Learning of programming languages: C, Perl
- Familiarity with:
    - TriCore architecture
    - Flash memory
    - Debugging (UDE PLS)
    - Synchronization techniques
    - Mutex techniques
    - Deadlocks
    - Executable file formats structure(HEX and ELF in particular)
    - Testing approaches
- Ability to generate professional documentation for each subproject, presenting and technical discussing with colleagues

[21] The test time is not exactly half as we have seen, because the overhead of multi-core operating system is not negligible

In conclusion the Beta release and verification of FTOS has brought several positive implications on Infineon microcontroller team:

- the code cloning allows to execute the new version of the FTOS
- multi-core is verified and the test time reduction is proven. This means that the testing time for the microcontroller team will be reduced resulting in a testing cost reduction
- multi-task hides an important issue and this has been identified, having a clear idea of the root cause

7.1 Next Steps

After the scheduler characterization, the thesis has shown that misalignments occur in the *Verify* code due to incorrect concept of the first alignment function. This has led to propose a new concept of the function. The next steps after this new concept proposal are:

- **New Alignment function implementation**: modify the first Alignment function by changing the condition (see [Figure 72]) and code the Barrier function
- **Define were the Barriers must be placed**: the flash memory is divided into levels and the Barrier function must be placed at any level end
- **Multi-task Verification**: the time slice must be varied and check if we obtain what we expect (see [Figure 59])

If the multi-task verification returns what we expect, the multi-task is verified and we can proceed with the Unit Testing. It is a type of white-box testing method and the source code is tested directly by checking if there are infinite loops, improper casting statements, undefined variables, etc. So, the goal of Unit Testing is to isolate each part of the code and show that the individual parts are correct. The approach adopted by Infineon to Unit Testing its software is the Test-Driven Development (TDD) for C language. It is an approach for building software incrementally that settles what tests must have done before the software is developed [30]. The test is small and followed immediately by the code to make that one test pass. No production code is written without first writing a failing unit test. TDD has its roots in Extreme Programming[22], but it is not a type of Extreme Programming. Normally, we design code, we write it and, then, we test (*Debug-Later Programming*) [30]. With TDD we use another way to solve programming problems. With the traditional mode of programming you can't estimate the time that will need for debugging and this could cause divergent debugging time. With TDD for embedded C, we can minimize the time spent to discover a bug and the time spent to find defect's root cause.

[22] Extreme Programming empowers the developers to respond to customer requirements changes, late in the software life cycle. Extreme programmers keep their design simple and clean. They get feedback by testing their software starting on day one. For more info see [35].

Appendix A

Below, there is the code of the Perl script used to fix and merge the HEX files. The numbers at left of the script indicates the line number; it is useful for explaining the code.

```
1  use FindBin;
2  use lib $FindBin::Bin;
3  use strict;
4  use Readonly;
5  use HEXParser;
6
7  Readonly my $BUS => 32;
8
9  my $lsl    = $ARGV[0];
10 my $dir    = $lsl . "\\Debug\\";
11 my $output = $dir . "\\Release\\";
12 my $nome_file    = $output . "\\nome_file.hex";
13
14 unless(-e $output or mkdir $output) {
15    die "Unable to create $output\n";
16 }
17
18 # read .lsl
19 open LSL, "<", $lsl . "\\nome_file.lsl";
20 my @rows = <LSL>;
21 close(LSL);
22
23 my $return;
24 opendir( DIR, $dir );
25 while ( my $file = readdir(DIR) ) {
26    next if ( ( $file eq '.' ) or ( $file eq '..' ) );
27    next if ( $file !~ m/nome_file_mpe_/xmsi );
28    next if ( $file !~ m/.hex/xmsi );
29    my $region = $file;
30    $region =~ s/nome_file_mpe_//xmsgi;
31    $region =~ s/.hex//xmsgi;
32
33    # get address of region based on .lsl
34    my ($index) = grep { $rows[$_] =~ m/memory\s$region\s/xmsi } 0 .. $#rows;
35    my $start = $rows[ $index + 6 ];
36
37    #   map (dest=bus:sri, dest_offset=hesadecimal_address, size=size_kB);
38    $start =~ m/dest_offset=(0x[0-9A-F].+)\,/xmsi;
39    $start = $1;
40    print qq/$region: $start\n/;
41
42    my $input = $dir . '\\' . $file;
43    my $hex   = HEXParser->new($BUS);
44    $return = $hex->read($input);
45    foreach my $address ( keys %{ $hex->{data} } ) {
46          my $value       = $hex->{data}->{$address};
47          my $new_address = hex( '0x' . $address ) + hex($start);
48          $hex->{data}->{ sprintf( "%08X", $new_address ) } = $value;
49          delete $hex->{data}->{$address};
50    }
51    $return = $hex->write( $output . $file );
```

```
52 }
53 closedir(DIR);
54
55 #create single file
56 open OUT, ">" . $nome_file;
57 opendir( DIR, $output );
58 while ( my $file = readdir(DIR) ) {
59    next if ( ( $file eq '.' ) or ( $file eq '..' ) );
60    next if ( $file !~ m/nome_file_mpe_/xmsi );
61    next if ( $file !~ m/.hex/xmsi );
62    open HEX, "<", $output . $file;
63    while (<HEX>) {
64          print OUT $_ if ( $_ !~ m/:00000001FF/xmsi );
65    }
66    close(HEX);
67 }
68 closedir(DIR);
69 print OUT qq/:00000001FF\n/;
70 close(OUT);
71
72 exit;
```

Perl is a high-level, general-purpose and interpreted programming language. An interpreter is a computer program that directly executes, i.e. no compiling mechanism. It is perfect to read and modify files, almost all types. There are many libraries available online, for each use. The list of `use` (from line 1 to 5) corresponds to the libraries declared in the script. The *readonly* is a facility for creating read-only scalars, arrays and hashes. From line 9 to 12, are defined inputs and outputs, which in this case are the files in the scheme [Figure 47]. From line 19 to 21, the LSL is converted to rows. The `my @rows = <LSL>` is used to construct a row object with the structure defined by LSL. The `my` keyword identifies a variable in Perl. In the slice of code beneath, is opened and read the directory DIR and is searched any file containing "`nome_file_mpe_`" and with "`.hex`" extension. The file name is the same of the region name in the LSL. The files names are extracted with regular expressions (*regex* aka); for more information about *regex* theory, read [31].

```
23 my $return;
24 opendir( DIR, $dir );
25 while ( my $file = readdir(DIR) ) {
26    next if ( ( $file eq '.' ) or ( $file eq '..' ) );
27    next if ( $file !~ m/nome_file_mpe_/xmsi );
28    next if ( $file !~ m/.hex/xmsi );
29    my $region = $file;
30    $region =~ s/nome_file_mpe_//xmsgi;
31    $region =~ s/.hex//xmsgi;
```

The 34-35 lines, extracts the correspondent start address from the LSL. Instead, from line 38 to 53 the destination offset is read from the LSL, is printed to the standard output the region with its start address, the HEX file is open and the destination offset is added to the incorrect address. So, `$new_address` is written on the correspondent HEX file and lastly the file is

closed. This is done for any HEX file created from TASKING. From line 56 to 70, any file is opened and printed without its *End Of File* (EOF) in a unique final HEX file.

Appendix B

The JLTE.pl (JAZZ Log To Excel) is a Perl script, that parses the JAZZ HTML log file, returns an Excel spreadsheet with all test times stacked and does some calculus. For the code of the script, see below.

```
1  use strict;
2  use warnings;
3  use Switch;
4  use Spreadsheet::WriteExcel;
5  use Time::HiRes qw(time);
6  my $start = time;
7  my $input = "datalog.html";
8  my $totaltesttime=0;
9  my $NumberTestTimes = 0;
10 my $numberv1s=0;
11 my $numberv0s=0;
12 my $numberv1p=0;
13 my $numberv0p=0;
14 my $timev1s=0;
15 my $timev0s=0;
16 my $timev1p=0;
17 my $timev0p=0;
18 my $testname='';
19 my $time=0;
20 my $tic=1/(100E6);
21 my $TIME_SLICE = 0;
22
23 my $workbook = Spreadsheet::WriteExcel->new('results.xls');
24 my $worksheet = $workbook->add_worksheet('Test Times');
25
26 my $title = $workbook->add_format();
27 $title->set_bold();
28 $title->set_align('left');
29 $title->set_color('white');
30 $title->set_bg_color('navy');
31
32 my $titleb = $workbook->add_format();
33 $titleb->set_bold();
34 $titleb->set_align('left');
35 $titleb->set_color('white');
36 $titleb->set_bg_color('green');
37
38 my $data = $workbook->add_format();
39 $data->set_align('left');
40
41 $worksheet->write_string(0, 0, 'TEST NAME', $title);
42 $worksheet->write_string(0, 1, 'TEST TIME [us]', $title);
43 $worksheet->write_string(0, 3, 'Total Test Time [us]', $titleb);
44 $worksheet->write_string(2, 3, 'Total Verify Time [us]', $titleb);
45 $worksheet->write_string(4, 3, 'Mean Verify Time [us]', $titleb);
46 $worksheet->write_string(6, 3, 'Time Slice [ms]', $titleb);
47
48 $worksheet->write_string(0, 5, 'v1s Time [us]', $title);
49 $worksheet->write_string(2, 5, 'v0s Time [us]', $title);
50 $worksheet->write_string(4, 5, 'v1p Time [us]', $title);
```

```
51 $worksheet->write_string(6, 5, 'v0p Time [us]', $title);
52
53
54 print "What is the value of TIME_SLICE?\n";
55 print "Write it with 0x at the beginning\n";
56 chomp ($TIME_SLICE = <>);
57 print "You choose $TIME_SLICE \n";
58
59 open (READ_HTML, $input) or die "could not open $input\n";
60 my $row = 0;
61
62 while(<READ_HTML>){
63
64    if ($_ =~ m/(Test_Time:[_]*)(\w*)/xmsi)
65
66          {
67                $totaltesttime = $totaltesttime + hex $2;
68                $NumberTestTimes++;
69                switch ($_){
70
71                            case m/v1s/xmsi   {$numberv1s++};
72                            case m/v0s/xmsi   {$numberv0s++};
73                            case m/v1p/xmsi   {$numberv1p++};
74                            case m/v0p/xmsi   {$numberv0p++};
75
76                                  }
77
78
79
80
82
83 #STAMP ONLY VERIFY TESTS
84
85  if($_=~m/(((\d*\s)*)(\w*)((v1s)|(v0s)|(v1p)|(v0p))([\w\[\]]*)
86  (((\s)*)(\d*)((\s)*)(Test_Time:)([_]*)(\w*)/xmsi)
87          {
88          $testname = $3.$4.$9;
89          $time = hex $17;
90          print "$testname --> $time ms\n";
91          $row++;
92          $worksheet->write_string($row, 0, $testname, $data);
93          $worksheet->write_number($row, 1, $time, $data);
94
95          switch ($_){
96
97                      case m/v1s/xmsi   {$timev1s = $timev1s + $time};
98                      case m/v0s/xmsi   {$timev0s = $timev0s + $time};
99                      case m/v1p/xmsi   {$timev1p = $timev1p + $time};
100                     case m/v0p/xmsi   {$timev0p = $timev0p + $time};
101
102                     }
103
104         }
105
106         }
107
108
109 }
110
```

```
111 print "\n";
112 print "Number of test times in JAZZ log: $NumberTestTimes \n";
113 print "\n";
114 print "Number of v1s test times in JAZZ log: $numberv1s \n";
115 print "\n";
116 print "Number of v0s test times in JAZZ log: $numberv0s \n";
117 print "\n";
118 print "Number of v1p test times in JAZZ log: $numberv1p \n";
119 print "\n";
120 print "Number of v0p test times in JAZZ log: $numberv0p \n";
121 print "\n";
122
123 my $totalverify=$timev1s + $timev0s + $timev1p + $timev0p;
124
125 $worksheet->write_number(1, 3, $totaltesttime, $data);
126 $worksheet->write_number(3, 3, $totalverify, $data);
127 $worksheet->write_number(5, 3, $totalverify/$row, $data);
128 $worksheet->write_number(7, 3, (hex $TIME_SLICE)*$tic*1E3 , $data);
129
130 $worksheet->write_number(1, 5, $timev1s, $data);
131 $worksheet->write_number(3, 5, $timev0s, $data);
132 $worksheet->write_number(5, 5, $timev1p, $data);
133 $worksheet->write_number(7, 5, $timev0p, $data);
134
135
136 close READ_HTML;
137 my $duration = (time - $start)*1000;
138 print "Execution time: $duration ms\n";
```

The list of *use* keywords imports the modules used in the script and *my* keywords defines the variables. When there are "*::*" means that we import a module that contains other modules. We have to take particular attention for "*Spreadsheet :: WriteExcel*"; this module allows for the creation of Excel binary files and permits the cell formatting, cell merges, multiple worksheets, formulae and also printer specifications. Here [32] is available the module and some explanations. From the line 23 to 51, we define the XLS file structure and the formats. There are defined 3 formats: `$title` and `$titleb` defines two types of titles (only the colors change) and `$data` that formats any data in the worksheet as left aligned. These formats are used in `$ worksheet -> write_string($row, $column, $string, $format)` that is a worksheet method. The `chomp` command at line 56 catches the time slice written by the user. When given no arguments to the `chomp`, the operation is performed on `$_`. From line 59 to 75, we open with `open (READ_HTML, $input)` the $input, that is the variable linked to the HTML log file. The `while(<READ_HTML>)` scrolls down all the lines of the HTML log. The if statement at line 64 permits to counts the *Verify* tests by selecting the line that both matches the *regex*

```
/(Test_Time:[_]*)(\w*)
```

and one of the others in the switch-case statement. From 111 to 121 we print the result on the standard output. The next if statement at lines 85-86 is more complex: it permits to parse and catch the test names and the relative test times with the commands

```
88          $testname = $3.$4.$9;
89          $time = hex $17;
```

that selects the correspondent round brackets in the *regex*. These names and data are then written with the commands in the lines 92-93. Finally from line 95 to 133 we parse and catch all the total test times for any type of verification. The `close READ_HTML` closes the file, after all the `while()` iterations because we have scrolled down all the file. The $duration variable only keeps control of the execution time. This simple script allows to greatly reduce the time of data copying and Excel calculations from the log results. With a *double-click*, you copy 105 tests and all the totals are calculated. This is critical because, for each time slice value, we have to follow the test procedure and therefore these 105 tests should be copied 5 times. The motivation is not just the time reduction, but also to limit the copying errors that can occur. Moreover, it is easily reusable in the future for other time measures.

Bibliography

[1] Infineon Technologies AG, "Thebe FTOSv3.x User Manual," Munich, 2010.

[2] Infineon Technologies AG, "System development handbook," Munich, 2007.

[3] R. Turner, "Toward Agile Systems Engineering Processes," April 2007.

[4] Altium, "TASKING Web Page," [Online]. Available: http://tasking.com/.

[5] Pls Development Tools, "UDE Web Page," [Online]. Available: http://www.pls-mc.com/universal-debug-engine-ude-and-debugger-for-aurix-tricore-power-architecture-cortex-mra_arm-7911-xe166xc2000-xscale-sh-2a_c166st10/universal_debug_engine-a-802.html.

[6] D. Attwell, "How is your brain powered?," European Research Council, April 2013. [Online]. Available: https://erc.europa.eu/erc-stories/how-your-brain-powered.

[7] P.Pavan, L.Larcher and A.Marmiroli, Floating Gate Devices: Operation and Compact Modeling, Kluwer Academic Publishers, 2004.

[8] P.Pavan, R.Bez, P.Olivo and E.Zanoni, "Flash Memory Cells — An Overview," IEEE, 1997.

[9] J.M. Rabaey, A. Chandrakasan, B. Nikolic, Digital Integrated Circuits, 2nd Edition, PEARSON, 2003.

[10] C. Preimesberger, "eWeek," 2011. [Online]. Available: http://www.eweek.com/c/a/Data-Storage/NAND-Flash-Memory-25-Years-of-Invention-Development-684048.

[11] Y. Pan, G. Dong and T. Zhang, "Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance," Rensselaer Polytechnic Institute, Troy, ALBANY, USA.

[12] O. Ginez, J.M. Daga, P. Girard, C. Landrault, S. Pravossoudovitch and A. Virazel , "Embedded Flash Testing: Overview and Perspective," Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier – Université de Montpellier II, Montpellier, 2006.

Bibliography

[13] Y. Cai, E.F. Haratsch, O. Mutlu and K. Mai, "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization and Analysis," Carnegie Mellon University, Pittsburgh, PA, 2012.

[14] S. Boboila and P. Desnoyers, "Write Endurance in Flash Drives: Measurements and Analysis," Northeastern University, Boston.

[15] International Organization for Standardization, "ISO 26262-3:2011(en) Road vehicles — Functional safety — Part 3: Concept phase," ISO, 2011. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:26262:-3:ed-1:v1:en.

[16] "AURIX™ Family – TC27xT," Infinoen Technologies AG, [Online]. Available: http://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-tm-microcontroller/aurix-tm-family/aurix-tm-family-%E2%80%93-tc27xt/channel.html?channel=db3a30433cfb5caa013d01df64d92edc.

[17] O. Pfeiffer and A. Ayre, "Using Flash Memory in Embedded Applications," Embedded System Academy, [Online]. Available: http://www.esacademy.com/en/library/technical-articles-and-documents/8051-programming/using-flash-memory-in-embedded-applications.html.

[18] "ARM Technical Support Knowledge Articles: HARVARD VS VON NEUMANN," ARM, 2011. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka11516.html.

[19] D.A. Patterson and J.L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, Revised Fourth Edition, Morgan Kaufmann, 2012.

[20] F. Mele, "Design and Development of a Multi-Task Multi-Core Operating System for Concurrent Test of Embedded Flash Memories on Automotive Microcontrollers," Sapienza - Università di Roma, Roma, 2015.

[21] R. A. Reid, "Task context switching RTOS - US7434222 B2," Infineon Technologies AG, October 2008. [Online]. Available: https://www.google.ch/patents/US7434222.

[22] A. S. Tanenbaum, Modern Operating System, 3 edition, Pearson Prentice Hall , 2009.

[23] A. M. McHoes and I. M. Flynn , Understanding Operating Systems - Sixth Edition, Course Technology, Cengage Learning, 2011.

[24] Shameem Akhter and Jason Roberts, Multi-Core Programming - Increasing Performance

through Software Multi-threading, Intel PRESS.

[25] E. Dijkstra, "Solution of a problem in concurrent programming control," Eindhoven University of Technology, Eindhoven, The Netherlands, September, 1965.

[26] A. Alexandrescu, "volatile: The Multithreaded Programmer's Best Friend," 01 February 2001. [Online]. Available: http://www.drdobbs.com/cpp/volatile-the-multithreaded-programmers-b/184403766.

[27] S. Zacchiroli, "Environnements et Outils de Développement, Cours 3 — The C build process," Laboratoire PPS, Université Paris Diderot - Paris 7, Paris, FRANCE, 2012.

[28] Altium BV, TASKING VX-toolset for TriCore User Guide, 2014.

[29] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, 9th Edition, WILEY, 2013.

[30] J. W. Grenning, Test-driven Development for Embedded C, The Pragmatic Programmers, 2011.

[31] J.E.Hopcroft - R.Motwani - J.D.Ullman, Introdution to Automata Theory, Languages and Computation, 3rd Edition, Addison Wesley, 2007.

[32] J. McNamara, "CPAN," 6 November 2013. [Online]. Available: http://search.cpan.org/~jmcnamara/Spreadsheet-WriteExcel-2.40/lib/Spreadsheet/WriteExcel.pm.

[33] Y. Gong, "Testing Flash Memories," 2004.

[34] Srinivas Nidhra and Jagruthi Dondeti, "BLACK BOX AND WHITE BOX TESTING TECHNIQUES –A LITERATURE REVIEW," IJESA, June 2012.

[35] D. Wells, "Extreme Programming: A gentle introduction," 2009. [Online]. Available: http://www.extremeprogramming.org/.