UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS num. 1981

Software and Hardware Architecture for Redundant Embedded Systems

Dino Šarić

Zagreb, September 2020.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Kako biste uklonili ovu stranicu, obrišite naredbu `\izvornik`.

Thanks to my parents for supporting me financially and giving me an opportunity of studying away from home. Thanks to my aunt Dijana and my friends for supporting me emotionally.

Thanks to my mentor izv. prof. dr. sc. Hrvoje Džapo and his assistant Ivan Pavić mag. ing. for friendly, fast and diligent approach during the studies and the writing of the thesis. Thanks to all the helpful current and former students from the college forums. Finally, thanks to the Youtubers that made understanding the college curriculum much easier.

Contents

List of Figures

List of Tables

Introduction

In a world with increasing number of electronic systems in hazardous environment, the correct operation of active systems is ever more important for ensuring less catastrophes. In year 2000, Air France Concorde flight crashed soon after its take-of killing 113 people, in 2005 Texas City refinery exploded killing 15 people and injuring 180. Similar disasters to these were the motivation for the creation of functional safety principles.

Figure 1: Texas refinery disaster

Figure 2: Air France Concorde disaster

Functional safety is the part of the overall safety that depends on a system or equipment operating correctly in response to its inputs.[6] In other words, the goal of functional safety is ensuring even when the system fails its response is predictable and safe. Today, the concept of functional safety is part of everyday life and applies to every industry one can think of. For example, functional safety ensures that airbags in a car instantly deploy during impact to protect the passengers. Another good example is an automated flight control system in the airplanes. Autopilot controls pitch and roll of the aircraft changing the heading and altitude, all of which is developed with respect to functional safety parameters, activating alarms and other measures when they are breached.[6]

Motivation of this paper is exploring how are principles of functional safety applied to the engineering projects. Investigate how and why redundancy is implemented in hardware and software. As a part of that, redundant microcontrollers are explored and compared to non-redundant counterparts. Additionally, functional safety additions to FreeRTOS operating system are implemented. Modifications add task replication and a option to measure execution time of tasks. Finally, secure bootloader is added,

bootloader has a command shell interface and has option of updating the current application.

The thesis is organized in the following way:

- chapter 1 gives brief introduction of functional safety process. Moreover, chapter gives a overview of how is hardware of embedded systems designed to support redundancy,
- chapter 2 investigates ARM Cortex R microcontroller's inner workings and what do they add over Cortex M,
- chapter 3 gives overview of of FreeRTOS and inner workings of tasks, scheduler and timers,
- chapter 4 gives overview of added safety functions to the FreeRTOS kernel,
- chapter 5 explains how the developed secure bootloader functions and its features and
- chapter 6 demonstrates the functionality of the developed software.

1. Functional safety in embedded systems

1.1. IEC standards

A variety of standards exist for specific or general markets, to facilitate the compliance of the systems to the functional safety principles. The standard IEC 61508 gives methods on how to apply, design, deploy and maintain automatic protection systems called safety-related systems. Along with the more general standard, industry specific standards exist:

- ISO 26262 for automotive passenger vehicles,
- IEC 61511 for the process industry and associated instrumentation,
- EN 50128 for software development of railway applications,
- IEC 61513 for nuclear power plants,
- IEC 62061 and ISO 13849 for machinery electrical control systems,
- IEC 62304 for medical systems and
- IEC 60730 for white goods.

Mentioned standards provide guidelines to assess rick and assign safety goals for safety-related systems of various industries. They provide frameworks for quantitative analysis of random failure rates and effectiveness of diagnostics to detect them. They also provide guidelines for maintenance of the safety-related systems after the deployment.

IEC 61508 is a umbrella functional safety standard applicable to all industries. It defines functional safety as: "part of the overall safety relating to the EUC (Equipment Under Control) and the EUC control system which depends on the correct functioning of the Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE) safety-related systems, other technology safety-related systems and external risk reduction facilities." The fundamental concept is that any safety-related system

must work correctly or fail in a predictable (safe) way. Functional safety relies only on active systems, and safety measures that rely on passive systems are not functional safety. [7]

1.2. Terminology

IEC 61508 standard introduces a lot of three letter acronyms. This section shines a light on the most common ones.

1.2.1. SIL

SIL or safety integrity level is a discrete level (one out of four) for specifying the safety integrity retirements of the safety instrumented functions (SIF) to be allocated to the safety instrumented system (SIS) e.i. SIL says how well the safety integrity function does its job. SIF and SIS are explained in subsection 1.2.2 and subsection 1.2.3 respectfully. SIL of 1 marks lowest safety integrity level and level 4 signalizes the highest. Figure 1.1 demonstrates that SIL is a function of frequency of a hazard and severity of the consequence.

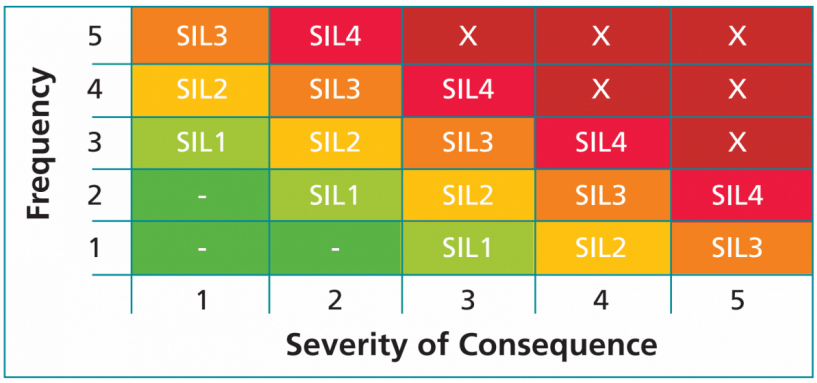

Figure 1.1: Generic SIL chart as a function of severity and frequency

1.2.2. SIF

Safety instrumented function (SIF) is a set of equipment used to implement an automatic protection function. Safety function shall have a specified SIL to achieve functional safety. Example of SIF is a number of sensors looking for the one **single** hazard scenario. Those sensors need to bring the system to the safe state if the hazard occurs. SIF are usually measured in how much risk can be removed. It is usually expressed in either in PFDaverage or in SIL.

1.2.3. SIS

SIS or safety instrumented system is a one or more safety functions put together to do the whole job. SIS can encompass multiple different functions and act in multiple different ways to overcome a number of hazard scenarios, as opposed to SIF that can only perform one action to overcome one hazard scenario.

1.3. Certification

For the certification, system integrity level is determined. SIL of a product is determined by three things[8]:

- the systematic capability rating,
- the architectural constrains for the element and
- the PFDaverage calculation.

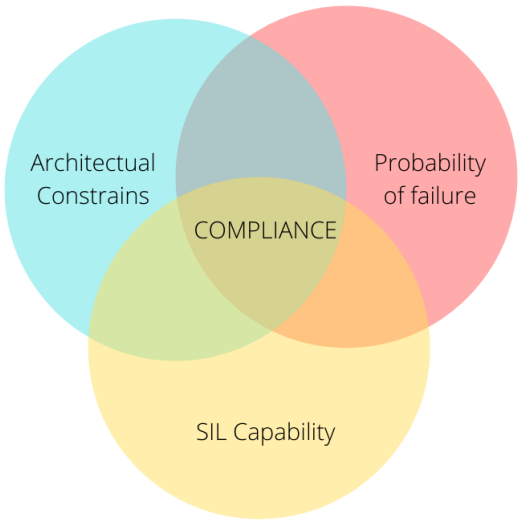

Figure 1.2: SIL requirements

Systematic Capability is achieved when the equipment used to implement any safety function is designed using procedures intended to prevent systematic design errors. The rigor of the required procedure is a function of a Safety Integrity Level (SIL). This is evaluated through an assessment of the quality management system for suppliers of process control and instrumentation for safety against the requirements in IEC 61508. If the QMS meets the requirements of 61508 a SIL Capability rating is issued.

Architectural constrains are estimated by the route 1H or route 2H as mentioned in IEC 61508. Route 1H is estimated by safe failure fraction(SFF). SFF is a percentage

of safe failures in all failures. Route 2H is an assessment of the reliability data for the entire element according to IEC 61508. It is mostly based around the historical data of a device and is based on a method that demands 90 percent confidence in the data. Instead of the SFF calculation, the Minimum Hardware Fault Tolerance is determined by a set of rules from IEC 61508.

Probability of Failure on Demand average (PFDaverage) is the probability that a system will fail dangerously, and not be able to perform its safety function when required. PFD can be determined as an average probability or maximum probability over a time period.

As mentioned before, a safety instrumented function needs to take risk out of the system. If there is no requirement to reduce risk than a function doesn't need a SIL rating or, for that matter, functional safety. When a function still needs to reduce risk after the process hazard assessment (PHA), SIL needs to be incorporated. Tables below give a link between SIL and expected safety.

PFD (probability of dangerous failure) and RRF (risk reduction factor) of low demand operation for different SILs as defined in IEC 61508 are shown in Table 1.1.

SIL	**PFD**	**PFD (power)**	**RRF**
1	0.1-0.01	$10^{-1} - 10^{-2}$	10-100
2	0.01-0.001	$10^{-2} - 10^{-3}$	100-1000
3	0.001-0.0001	$10^{-3} - 10^{-4}$	1000-10,000
4	0.0001-0.00001	$10^{-4} - 10^{-5}$	10,000-100,000

Table 1.1: Link between SIL, PFH and RRF for low demand operation

For continuous operation, Table 1.2 is provided (PFH - Probability of dangerous failure per hour).

SIL	**PFH**	**PFH (power)**	**RRF**
1	0.00001-0.000001	$10^{-5} - 10^{-6}$	100,000-1,000,000
2	0.000001-0.0000001	$10^{-6} - 10^{-7}$	1,000,000-10,000,000
3	0.0000001-0.00000001	$10^{-7} - 10^{-8}$	10,000,000-100,000,000
4	0.00000001-0.000000001	$10^{-8} - 10^{-9}$	100,000,000-1,000,000,000

Table 1.2: Link between SIL, PFH and RRF for continuous operation

When a safety instrumented function fulfills aforementioned SIL requirements a certificated can be assigned for the SIL. Example of a certificate is given in Figure 1.3.

Certificate

Functional Safety

TÜVRheinland CERTIFIED

www.tuv.com
ID 0600000000

No.: 968/FSP 1503.00/18

Product tested	General purpose microprocessor design including safety features	**Certificate holder**	ARM Ltd. 110 Fulbourn Road Cherry Hinton CB1 9NJ Cambridge United Kingdom
Type designation	ARM Cortex-R5 Processor Revision r1p3		
Codes and standards	IEC 61508 Parts 1-7:2010 (in extracts)		ISO 26262 Parts 1-10:2011 (in extracts)
Intended application	The ARM Cortex-R5 Processor complies with the requirements of IEC 61508 for SIL 3 regarding the avoidance of systematic faults for a Compliant Item and complies with the requirements of ISO 26262 for ASIL D regarding the avoidance of systematic faults for a Safety Element out of Context (SEooC). As a result of this the Cortex-R5 Processor can be used in safety-related applications up to SIL 3 according IEC 61508 and up to ASIL D according to ISO 26262.		
Specific requirements	The requirements and constraints mentioned in the Cortex-R5 Safety Manual have to be taken into account by the user.		

Valid until 2023-01-19

The issue of this certificate is based upon an examination, whose results are documented in Report No. 968/FSP 1503.00/18 dated 2018-01-19.
This certificate is valid only for products which are identical with the product tested.

Köln, 2018-01-19

TÜV Rheinland Industrie Service GmbH
Bereich Automation
Funktionale Sicherheit
Am Grauen Stein, 51105 Köln
Certification Body Safety & Security for Automation & Grid

Dipl.-Ing. Thomas Steffens

www.fs-products.com
www.tuv.com

DAkkS Deutsche Akkreditierungsstelle D-ZE-11052-02-00

TÜVRheinland®
Precisely Right.

10/222 12.12 E A4 ® TÜV, TUEV and TUV are registered trademarks. Utilisation and application requires prior approval.

TÜV Rheinland Industrie Service GmbH, Am Grauen Stein, 51105 Köln / Germany
Tel.: +49 221 806-1790, Fax: +49 221 806-1539, E-Mail: industrie-service@de.tuv.com

Figure 1.3: ARM Cortex-R5 Processor SIL 3 certificate

1.4. Embedded processors redundancy - lockstep

1.4.1. Introduction

Consider a system with one processing unit. Such system is obviously non-redundant from the aspect of instruction execution. Lockstep is redundancy mechanism for increasing the system reliability by introducing at least one redundant processing unit. Redundant processing unit replicates the behavior of the original processing unit. Depending on number of processing units lockstep can provide fault-detection, if there are less than three processing units in a system, or both fault-detection and fault-tolerance with more than two processing units in a system. [15] Figure 1.4 shows CPU1 and CPU2 in lockstep. CCU or CPU compare unit compares the outputs from the CPUs and throws an error if there is a mismatch.

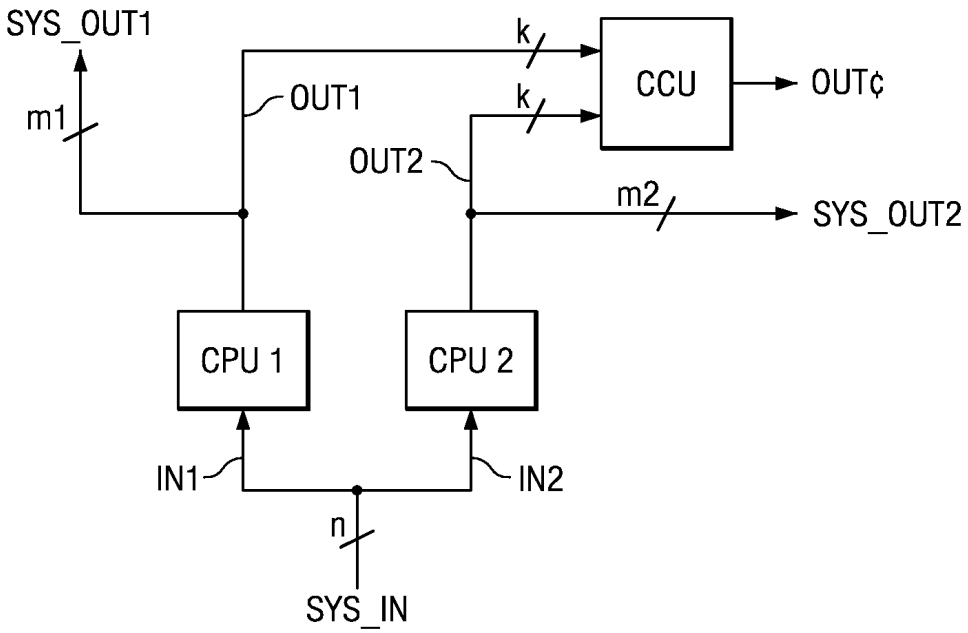

Figure 1.4: Block diagram of two processors in lockstep

One approach in categorizing errors detectable by lockstep is soft and hard errors. Soft errors describe errors that happen by temporary anomaly e.g. electrostatic discharge (ESD), radioactive decay, cosmic radiation. These errors lead to unintended transient signals or conditions in CPU or in peripherals. Their lifetime is usually 2 milliseconds or less. This kind of errors can affect an instruction execution or computation, but the next time that the same operation will be performed the error will not be reproducible. On the other hand, hard errors are permanent and are caused by a long term degradation of the silicon/logic over time. Hard errors may also oc-

cur by corrupted memory cells or other circuit components due to ionizing radiation, manufacturing inconsistencies, or exposure to high current which may cause metal migration. Generally, they are caused by physical defect of the hardware and they are not recoverable.

Lockstep architecture can detect the aforementioned soft and hard faults. If three processors are lockestepped, they are also fault-tolerant. [9]

Locksteped processors can get upto SIL3, but not SIL4 as a use of on-chip redundancy is limited by IEC 61508 standard to SIL3. Therefore, although it may seem, lockstep is certainly not panacea in all safety systems and certain amount of work is needed to justify its usage. Nevertheless, for all safety integrity levels, but SIL4 lockstep is often used extensively (e.g. automotive applications). [15]

1.4.2. Delayed lockstep [10]

To battle the transient errors (soft errors) e.g. cosmic radiation that lead the CPU or peripherals into unintended states delayed lockstep was invented by Texas Instruments. One of the CPUs' input is delayed, while other one's output is delayed ensuring that if a transient error happens it will not affect both CPUs on the same instructions. Figure 1.5 visualizes the delayed lockstep.

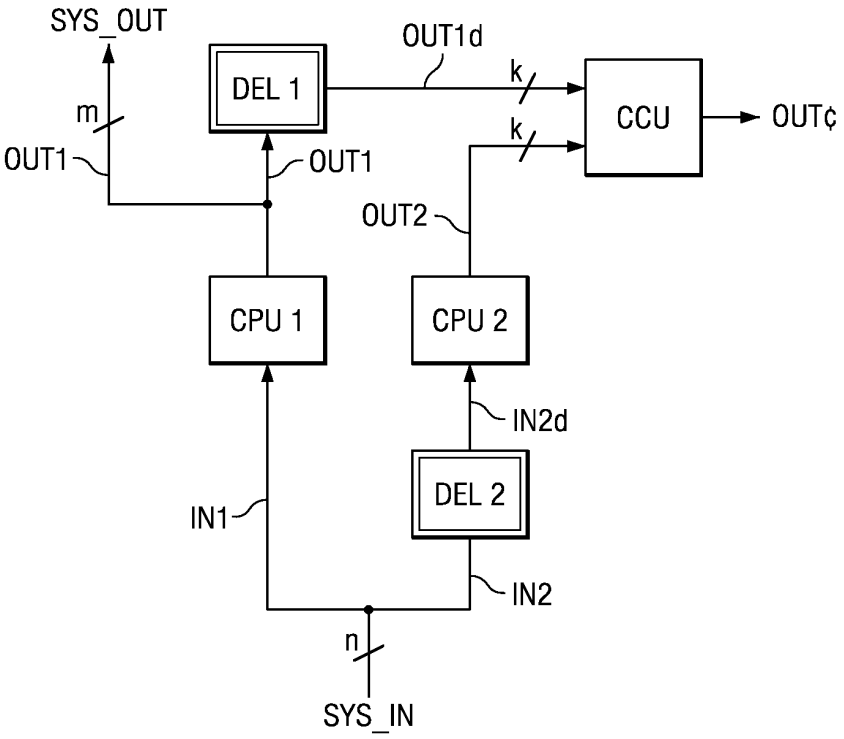

Figure 1.5: Block diagram of two processors in delayed lockstep

1.4.3. Positional diversity

When designing lockstep architecture positional diversity shall also be considered. Two cores shall be rotated 90 degrees and flipped in relation to each other. Atleast 100 um space is left in between them. Physical diversity gives some assurance of a different failure mechanism if there is a manufacturing flaw causing physical damage to the die. Figure 1.6 visualizes an example of two cores on the silicon. ž

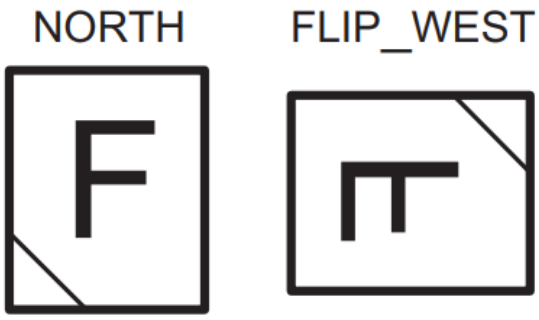

Figure 1.6: Typical orientation of the cores [12]

The common cause failures that physical diversity is called to restrict might be due to manufacturing flaws or even susceptibility to radiation due to the direction of some of the components used in the core design. A simple example for a potential manufacturing flaw that might be common to both cores without the physical diversity, suppose that there is a geometry of a certain component that was prone to collecting moisture and it was processed in a way that this component was traveling parallel to some chemical spray in the process. This might lead to trapped materials at this component in the silicon but if the components were rotated and flipped it would make the likelihood of trapping contaminates less likely. In this way, making the two cores physically diverse can keep both cores from having the same defective component. [9]

1.4.4. Debugging

During a debug session, where the execution of the code is halted, several asynchronous halting events occur, with a possibility to lead to a loss of Lockstep (indication of non-existing errors). For this reason, when halting debug events are detected, the compare unit is automatically disabled. In order to set the Lockstep mode on and the compare unit in operational state a CPU reset is required. [13]

1.4.5. Example use case [9]

A simplified example application is covered in this subsection to better explain the benefits of the lockstep. It should be noted that without a prior functional-hazard analysis this is not an optimal use case.

Beside the fact that the rate of aircraft engine failures have dramatically decreased, at the point that flight crews is most likely that will not ever meet one in their whole career, it still remains a possibility. Because of this infrequent occurrence of an engine failure the crew is not always able to identify and to handle such a malfunction. As a result, erroneous operations of the crew in such a case could lead to devastating consequences for them and for the passengers.

The likely most complex and crucial component on an airplane is the engine. Main part of the safety relies on this component. In this example we will focus on the fire detection scenario of the engine. In avionics, there is the principle of keeping the aircraft trajectory as the highest priority duty [9]. Thus, in case of an engine malfunction, the system should stabilize the aircraft trajectory first, before proceeding with further actions to resolve the engine problem. This could probably cause a larger damage to the engine but it would help to ensure the safety of humans and the aircraft itself.

One of the functions of the engine's single-core MCU is to receive the raw values from temperature sensors, making sense of them and notifying the crew if the temperature is too high to prevent fire. After taking the appropriate actions to stabilize the aircraft trajectory (i.e activate a redundant engine) the system sets the defective engine to idle state. In high altitude the level of radiation that occurs from cosmic rays is significantly high. Such a radiation can cause a bit flip in the core of the system that executes this temperature calculation. If the bit flip happens during the conversion of the raw values an spurious alarm could be raised. The erroneous calculation could end up in a non-realistic over-limit temperature value, leading to an undesirable deactivation of the engine and getting the flight crew into trouble. One can conclude, Single-core MCU can not detect an error in context of executing the commands.

If in place of a single-core MCU, a lock-step MCU with two cores is placed such error would be detected and could temporarily set the system in safe-state. The two cores operating in Lockstep, most likely would not experience the same bit flip and they would produce divergent outputs. This discrepancy in the comparison of the output signals would generate an error.

From the error detection, implementation differs according to the system requirements. For instance, if the system could handle to "wait" the time that is required for a soft-reset of the MCU, it could be determined if the error is temporal or permanent.

In case of permanent error the system could finally set the engine idle and enable a redundant one. In case of a soft error, a reset of the system could be enough to continue operating efficiently.

2. ARM Cortex-R functional safety additions over ARM cortex-M

2.1. ARM Cortex-R introduction

The ARM Cortex-R is a family of 32-bit RISC processors. The cores are optimized for hard real-time and safety-critical applications. Cortex-R family consists of: ARM Cortex-R4(F), ARM Cortex-R5(F), ARM Cortex-R7(F), ARM Cortex-R8(F and ARM Cortex-R52(F).

The Cortex-R is suitable for use in computer-controlled systems where very low latency and/or a high level of safety is required. An example of a hard real-time, safety critical application would be a modern electronic braking system in an automobile. The system not only needs to be fast and responsive to a plethora of sensor data input, but is also responsible for human safety. A failure of such a system could lead to severe injury or loss of life. Other examples of hard real-time and safety critical applications include medical devices, electronic control units (ECU), robotics etc.

2.2. ARM Cortex M introduction

The ARM Cortex M is also a family of 32-bit RISC processors. The cores are optimized for low-cost and energy-efficient microcontrollers. These cores are being used in tens of billions of consumer devices. The processors are intended for deeply embedded applications that require fast interrupt response features [3]. The family consists of: Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, Cortex-M4, Cortex-M7, Cortex-M23, Cortex-M33, Cortex-M35P and Cortex-M55.

2.3. ARM Cortex-R processor lockstep

2.3.1. Dual core lockstep

A Cortex-R5 processor group has four configurations, as described in [1]:

- single CPU,
- twin CPU,
- redundant CPU and
- split/lock configuration.

Twin CPU configuration includes two individual and decoupled CPUs. Each CPU has its own cache RAMs, debug logic and bus interfaces to the rest of the SoC. It offers higher performance than a standard single CPU configuration.

In redundant CPU configuration (lockstep), there is a functional CPU and a second redundant copy of the majority of the CPU logic. The redundant logic is driven by the same inputs as the functional logic. In particular, the redundant CPU logic shares the same cache RAMs as the functional CPU. Therefore only one set of cache RAMs is required. The redundant logic operates in lock-step with the CPU, but does not directly affect the processor behavior in any way [1]. The CPU outputs to the cache RAMs are driven **exclusively** by the functional CPU. The comparison logic for comparing the outputs of the redundant logic and the functional logic can detect a single fault that occurs in either set of logic. ARM provides example comparison logic, but the developer can change it during the implementation.

Split/lock configuration is a combination of two previously mentioned configurations. This mode includes two processors. If a processor is in split mode the CPUs work in twin CPU configuration, and if the processor is in lock mode it is in redundant configuration.

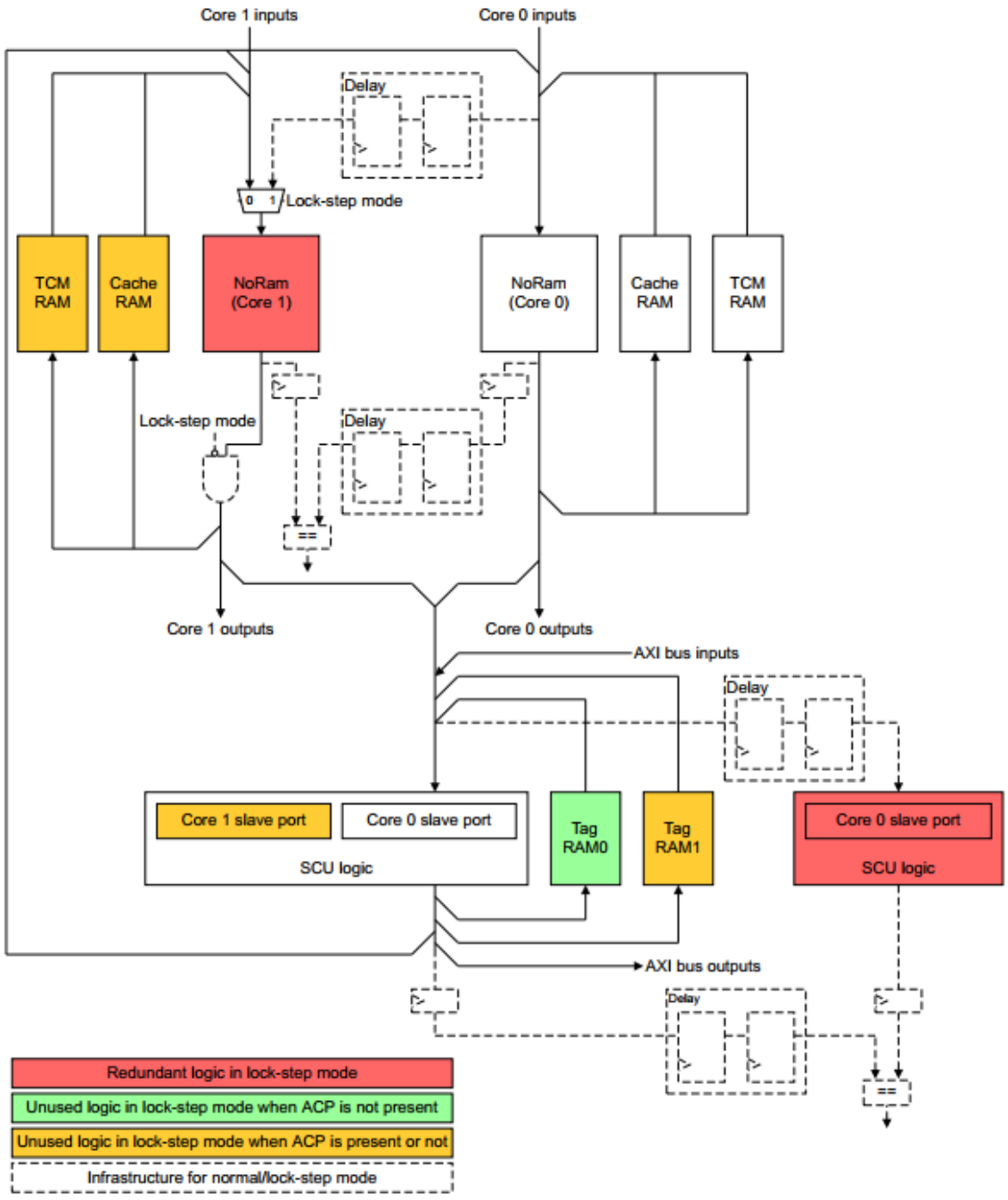

Figure 2.1: Split/lock configuration [2]

2.3.2. Triple core lockstep

ARM triple core lockstep architecture (TCLS) builds up on the industry success of the ARM Cortex-R5 dual-core lock-step (DCLS). The TCLS architecture adds a third redundant CPU unit to the DCLS Cortex-R5 system to achieve fail functional capabilities and hence increase the availability of the system [16].

Cores in triple lockstep have shared data and instruction cache, but each Cortex-R5 has its own clock tree. Figure 2.2 shows system level solution to mitigate soft

errors occurring in the redundant CPUs. On the right side of the figure TCLS assist unit is visualized. TCLS assist unit supports the lockstep functioning of the CPUs and handles the error recovery process. The unit consists of a majority voter, error detection logic and synchronization logic.

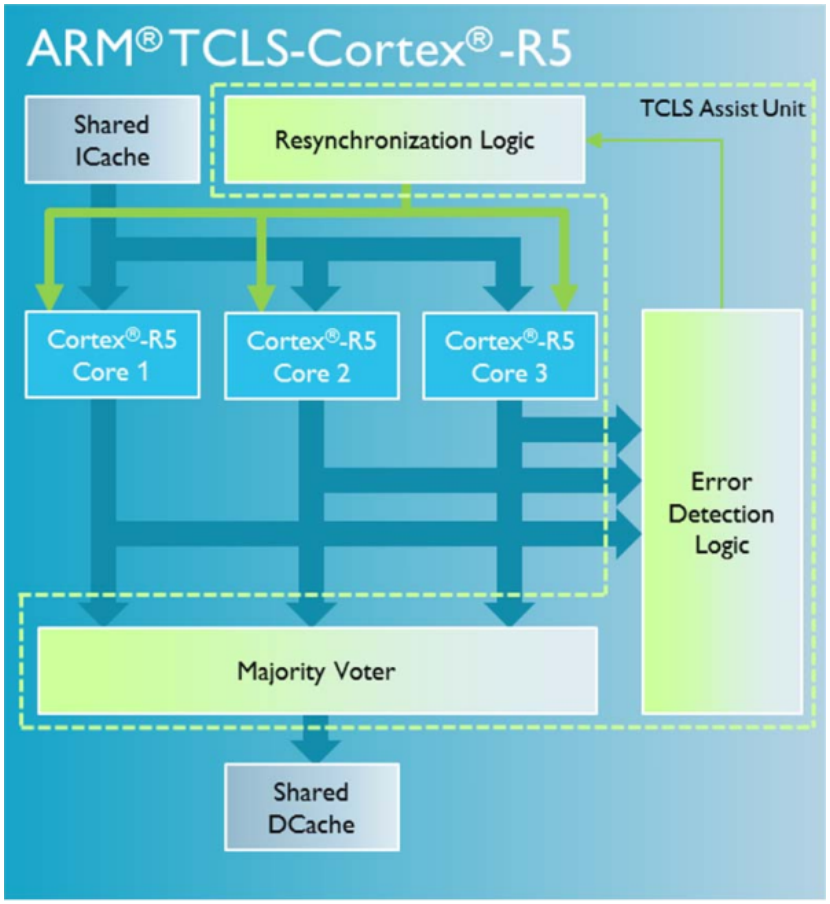

Figure 2.2: ARM triple core lockstep of Cortex-R5 [16]

At every clock cycle, the instructions to execute are read from the shared instruction cache or TCM (tightly coupled memory) and distributed to the triplicated CPUs. The outputs from the CPUs are majority voted and forwarded to the shared data cache, TCM, and I/O ports. Simultaneously, the Error Detection logic checks if there is any mismatch in the outputs delivered by the three CPUs. If there is a mismatch, all CPUs are interrupted and the Error Detection logic identifies whether it is a correctable error (i.e., only one of the CPUs delivers a different set of outputs) or an uncorrectable one (i.e., all CPUs deliver different outputs). If the error is correctable, the TCLS passes the control to the resynchronization logic to correct the architectural state of the erroneous CPU, that is, to resynchronize all the CPUs. Note here that the Majority Voter acts as an error propagation boundary, preventing correctable errors from propagating to memories and I/O ports. In the highly unlikely case that the error is uncorrectable, the TCLS transitions to a fail-safe operation state [16].

As mentioned in last paragraph, majority voter is in the critical path of the system, but the error detection logic is out of the critical path and is pipelined to increase performance.

Figure 2.3 show the flow diagram of the resynchronization logic. Upon a correctable error is detected, the resynchronization logic can immediately trigger the CPU resynchronization process. This action prevents the interruption of the critical real time tasks. It should be noted that the system can continue working on two remaining CPUs, which are in a functionally correct state. The third CPU is recovered from the two functioning CPUs. Unlike the dual core lock step the recovery of the triple core lockstep is automatic and transparent to the software [16].

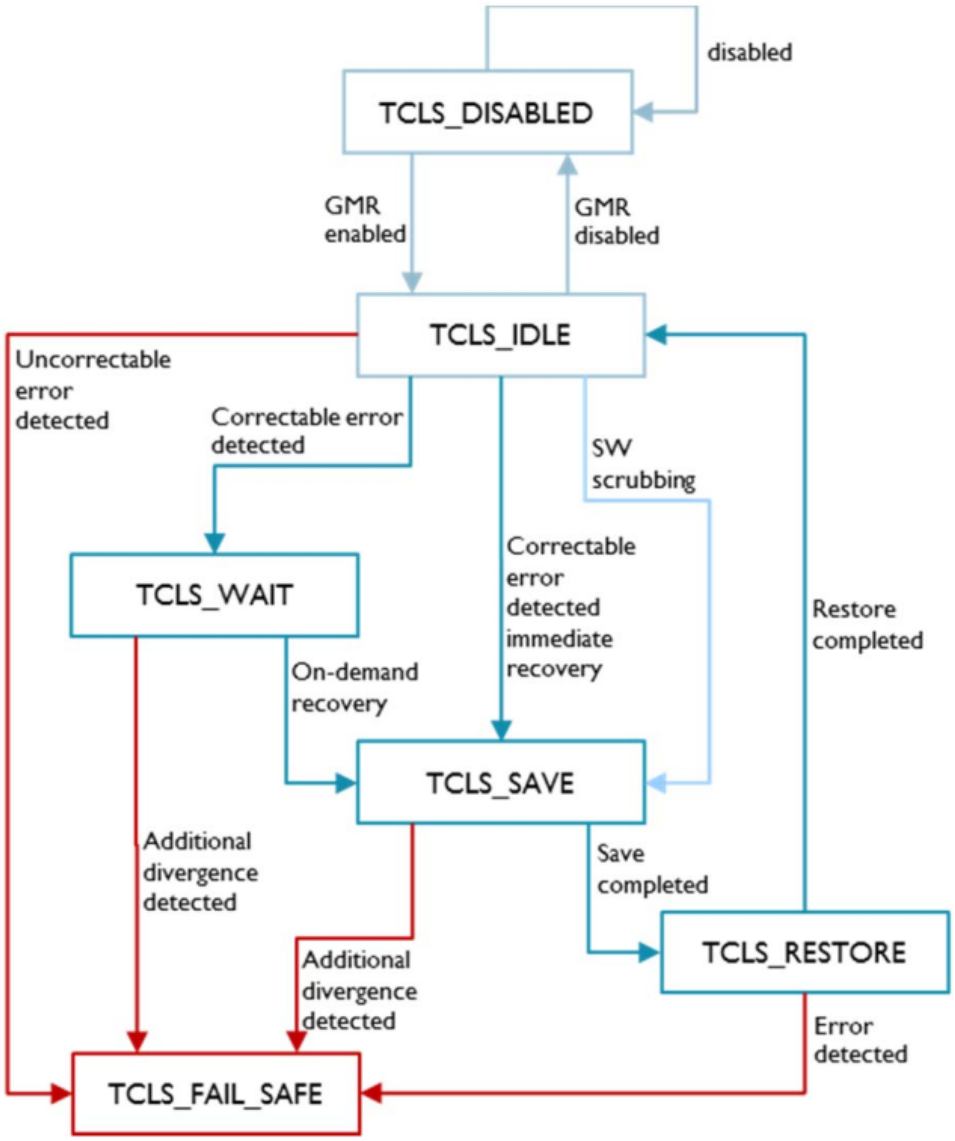

Figure 2.3: TCLS resynchronization finite state machine [16]

2.4. RAM error correction

In microcontrollers, stray radiation and other effects can cause the data stored in RAM (random access memory) to be corrupted (bit flip). The tightly coupled memories (TCMs) and caches on a Cortex-R processor can be configured to detect and correct errors that can occur in the RAMs. Extra, redundant data (checksum) is computed by the processor and stored in RAMs alongside the real data. When the processor reads the data from the RAM, it checks the redundant data matches the real data and can either signal an error, or attempt to correct the data.

Cortex-R5 allows the usage of:

- parity,
- 64-bit ECC or
- 32-bit ECC [1].

With parity an error can be detected, but it can't be corrected. Using either 64-bit or 32-bit ECC (error checking and correction) allows to detect up to two errors in a data chunk (either 64-bit or 32-bit) and correct any single error in the data chunk. Parity adds one redundant bit for every byte. 64-bit ECC adds one byte per 8 byte data chunk. 32-bit ECC adds seven bits per 4 bytes of data chunk.

Cortex-R8 uses 32-bit, 34-bit or 64-bit ECC of variable data chunks to protect its RAMs [2].

2.5. Interrupt handling

According to [1], Interrupt handling in the processor is compatible with previous ARM architectures, but has several additional features to improve interrupt performance for real-time applications.

ARM Cortex-R5 connects the vectored interrupt controller (VIC) over a port, unlike ARM Cortex-M series microcontrollers that only has VIC close to the core. Having a VIC over the port provides faster interrupt entry, but one can disable it for compatibility with earlier interrupt controllers [1]. ARM Cortex-R5 doesn't allow tail chaining of the interrupts, nor handling of late arriving interrupts, unlike ARM Cortex-M series.

Both ARM Cortex-R and Cortex-M series microcontrollers have an ability to abandon assembler instructions LDM and STM to lower the interrupt's latency. While Cortex-R microcontrollers will rerun the whole command upon the return from the interrupt, the ARM Cortex-M microcontrollers will just continue where they left of.

2.6. CPU Compare Module for Cortex-R5F from Texas Instruments

CPU compare module (CCM) detects run-time faults in devices as the CPU and VIM (vectored interrupt controller module) and forwards them to the next step in error handling of the microcontroller i.e. to ESM (error signaling module). Alongside the run-time fault testing, CCM incorporates a self-test capaility to allow for boot time checking of hardware faults within the CCM itself [13].

Main features of the CCM are:

- run-time detection of faults,
- self-test capability and
- error forcing capability.

There are four modes of diagnostics for CPU/VIM output compare.

- active compare lockstep,
- self-test,
- error forcing and
- self-test error forcing mode.

Active compare lockstep mode is default on start-up. The bus output signals of both CPU and VIMs are compared. List of all compared output signals is available in [13, p. 500].

In self-test mode, test patterns are automatically generated by the CCM-R4F to determine its correct functionality. In case of an error detection that indicates a hardware fault on the module itself, a "CCM-R4F - selftest" flag will be raised to the ESM. After the completion or termination of the self-test, if no error occurred, the self-test complete flag is set to notify the system to proceed appropriately. The compare match test and compare mismatch test are generated to ensure proper functioning. In compare match test an identical vector is applied to both input ports at the same time expecting a compare match. Compare mismatch test is is similar to the compare match test, but one of the input vectors has one bit flipped. On the output, compare mismatch is expected.

Error forcing mode ensures that an error on the CCM's input is detected. In the case when the error from error forcing mode is not detected there is a hardware failure present. The difference between compare missmatch test and error forcing mode is that error forcing mode ensures that compare error output signal actually asserts. This mode lasts for one CPU cycle.

3. FreeRTOS kernel

3.1. Introduction

FreeRTOS is a real-time operating system kernel (RTOS) for embedded devices. It has been ported to 35 microcontroller pletforms and has MIT open source licence hence the free in the name. [17]

Operating system (OS) is designed to be small and simple. The kernel itself has only three C files. It is written mostly in C with some assembly code included for the scheduler routines.

FreeRTOS is ideally suited for deeply embedded real-time applications that use microcontrollers or small microprocessors. This type of application normally includes a mix of both hard and soft real-time requirements. [5]

Soft real-time requirements are those that state a time deadline—but breaching the deadline would not render the system useless. For example, responding to keystrokes too slowly might make a system seem annoyingly unresponsive without actually making it unusable. [5]

Hard real-time requirements are those that state a time deadline—and breaching the deadline would result in absolute failure of the system. For example, a driver's airbag has the potential to do more harm than good if it responded to crash sensor inputs too slowly. [5]

FreeRTOS features:

- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- Flexible, fast and light weight task notification mechanism
- Queues
- Binary and counting semaphores
- Mutexes
- Recursive mutexes

- Software timers
- Event groups
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace recording
- Task run-time statistics gathering
- Optional comemercial licensing and support
- Full interrupt nesting model (on some architectures)
- Tick-less capability for extreme low power applications
- Software manages interrrupt stack when appropriate (could save RAM space)

FreeRTOS file structure is shown in Figure 3.1.

```
Source
├── include
│   ├── FreeRTOS.h
│   ├── list.h - Primary structure used inside the kernel
│   ├── message_buffer.h
│   ├── portable.h
│   ├── projdefs.h
│   ├── queue.h
│   ├── semphr.h - Semaphores
│   ├── stream_buffer.h
│   ├── task.h
│   └── timers.h - Software timers
├── portable
│   ├── [compiler] e.g.  GCC
│   │   └── [architecture] e.g.  ARM_CM4F
│   │       ├── port.c - Architecture and compiler specific
│   │       └── portmacro.c
│   └── MemMang
│       └── heap_X.c - X is a heap number used
├── croutine.c
├── event_groups.c
├── list.c
├── queue.c
├── stream_buffer.c
├── tasks.c - Tasks and scheduler implementation
└── timers.c
```

Figure 3.1: FreeRTOS file structure

3.2. Inner workings of the tasks

Every FreeRTOS task has a stack and a task control block, or short TCB. Kernel uses the TCB to manage tasks. A TCB contains all information necessary to completely describe the state of a task. [11] A FreeRTOS task can exist in five states: running, blocked, ready, suspended and deleted. A state diagram is shown in Figure 3.2.

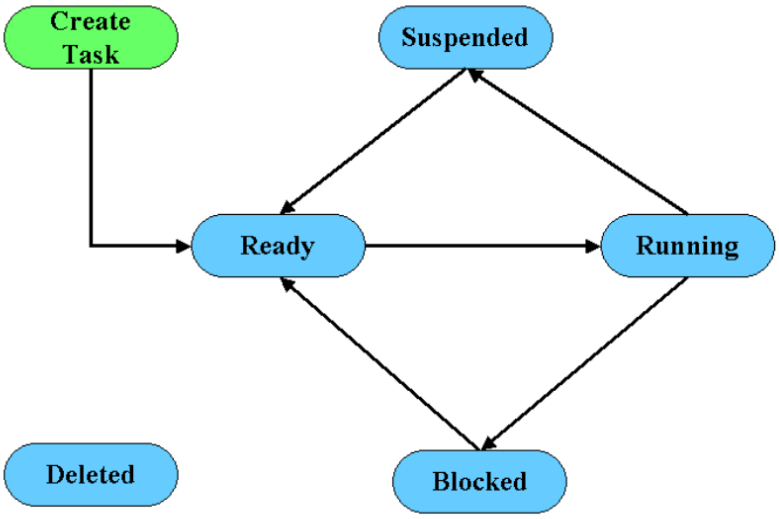

Figure 3.2: FreeRTOS task states[11, p 10]

When a new task is created its TCB is populated. New tasks are immediately placed in a ready list. Whole scheduling is comprised of a lot of lists.

Ready list is arranged in order of priority with tasks of equal priority being serviced round robin. Ready list is not actually a single list, rather a configMAX_PRIORITIES number of lists. Each priority level has a list for it. When scheduler looks for the next task it walks from the tasks with highest priority to the one with the lowest. Variable pxCurrentTCB points to a process in the ready list that is currently running.

The tasks in FreeRTOS can be blocked when accessing a resource that is not currently available. The scheduler blocks the tasks when they attempt to read from an empty container or write into a full one. This is also true for the semaphores, as they are a queue of size one in the background.

As indicated earlier, access attempts against queues can be blocking or non-blocking. The distinction is made via the xTicksToWait variable which is passed into the queue access request as an argument. If xTicksToWait is 0, and the queue is empty/full, the task does not block. Otherwise, the task will block for a period of xTicksToWait scheduler ticks or until an event on the queue frees up the resource.

Tasks can also be blocked without a use of containers. FreeRTOS provied vTaskDelay and vTaskDelayUntil functions for this purpose. When a task is delayed it is put onto a delay list. On every tick, scheduler checks if one of the tasks from the delay lists are unblocked. If they are, they are moved to the ready list.

Any task or, in fact, all tasks except the one currently running (and those servicing ISRs) can be placed in the Suspended state indefinitely. Tasks that are placed in this state are not waiting on events and do not consume any resource or kernel attention until they are moved out of the Suspended state. When unsuspended, they are returned to the Ready state.

Finally, tasks can also be deleted. When delete is requested task is put in a deleted state. Deleted state is required because tasks are not deleted immediately after the call. Rather tasks are deleted, and its resources released, from the IDLE task. IDLE task has the lowest possible priority so this job may take some time.

3.3. Inner workings of the scheduler

This section gives a brief overview of a FreeRTOS scheduler.

Figure 3.3 shows an overview of the scheduler algorithm. The scheduler operates as a timer interrupt service routine that is called once every tick. Tick period is defined by configTICK_RATE_HZ.

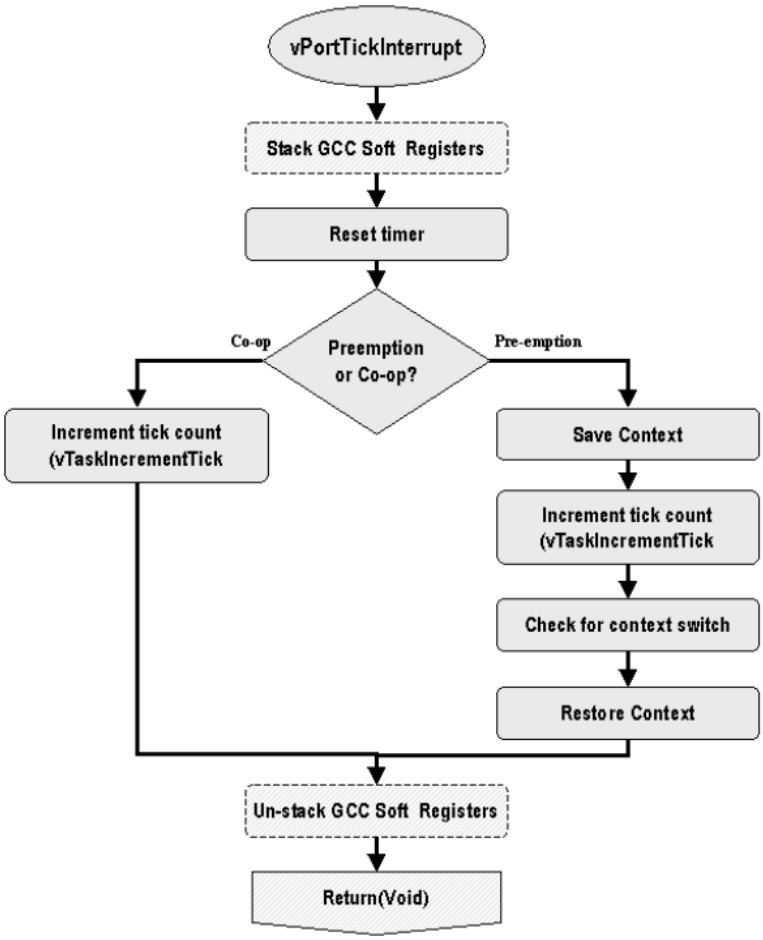

Figure 3.3: Scheduler algorithm[11, p 20]

Context saving is done for the current task. Needed registers are saved on top of the task's stack. It is worth noting, when a task is first created its task is artificially filled. After saving the context scheduler increments the tick and checks if any other task with higher priority has been unblocked, or there is a task with same priority ready. Finally, context is restored and scheduler returns from the interrupt.

Figure 3.4 shows the algorithm for vTaskIncrementTisk. vTaskIncrementTick is called once each clock tick by the HAL (whenever the timer ISR occurs). The right hand branch of the algorithm deals with normal scheduler operation while the left hand branch executes when the scheduler is suspended. As discussed earlier, the right hand branch simply increments the tick count and then checks to see if the clock has overflowed. If that's the case, then the DelayedTask and OverflowDelayedTask list pointers are swapped and a global counter tracking the number of overflows is incremented. An increase in the tick count may have caused a delayed task to wake so check is performed.

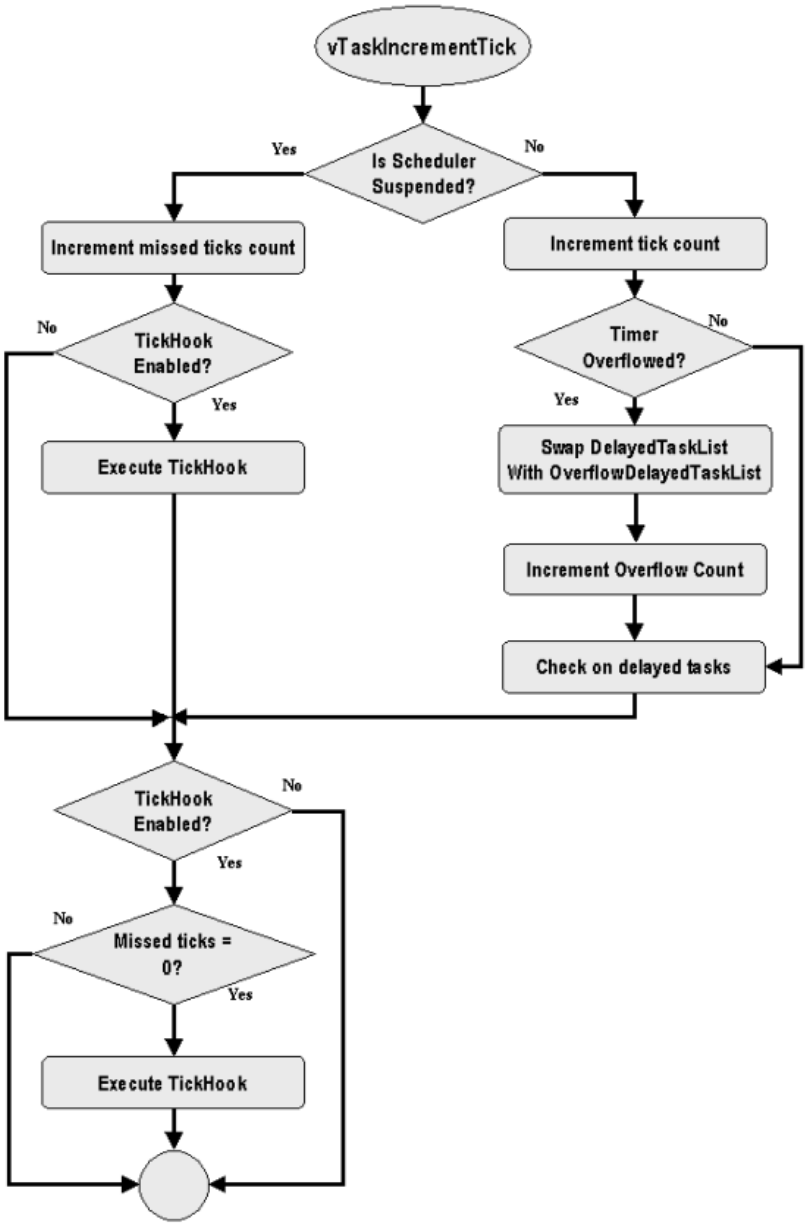

Figure 3.4: vTaskIncrementTick algorithm[11, p 31]

More about the scheduler can be found in [11].

3.4. Inner workings of the timers

Similarly to tasks, FreeRTOS timers have a control block. Timer's control block contains timers period, name, does it auto reload and list item. Active timers are stored in current timer list in order of expiry time, first element is the one that will expire first.

When timers are included from the configuration, scheduler on start up creates the

timer daemon service (its priority is modifiable with configTIMER_TASK_PRIORITY). Timer daemon has a job of processing the expired timers and receiving the commands. All commands controlling the timers the commands are not sent directly to the requested timer, rather all commands are sent to the queue to be later processed by the timer's daemon.

Timer daemon normally just waits for unblocking of the next timer or a new commands from the queue. When a timer expires daemon is woken up, it processes the timer, checks again for received commands and goes back to waiting. If a new commands has arrived the same is expected, first the commands is processed than the task goes back to waiting.

4. FreeRTOS functional safety additions

4.1. Timed tasks addition

4.1.1. Introduction

Atop of all FreeRTOS functionality ability of measuring task's run time and its total time (running + other states) is not provided. Such feature is extremely useful for hard real-time embedded systems. In such systems breaching the deadline means failure of the system.

Adding the aforementioned timers to the tasks means ability to detect if the task ran for too long or it didn't get enough processor time. When such event has occurred it can be properly handled. For example, for an embedded system that is periodically polling a speed of rotation of an engine if it isn't polled in time it can raise an alarm to force the reading.

4.1.2. Architecture

When timed tasks are created two software timers are created tied to it. One is called overrun timer and is used to detect when the task is in a running state longer than defined. Second, overflow timer is used to detect if the task is running properly asynchronously i.e. timer doesn't stop ticking even if the timed task is inactive.

The overflow timer is started when the timed task is switched into the first time. It is started from the context switch. Code section is shown in Figure 4.1. Function prvStartOverflowTimer checks if the timer is started and if it isn't it starts the timer otherwise it doesn't do anything.

```
3224        /* Check for stack overflow, if configured. */
3225        taskCHECK_FOR_STACK_OVERFLOW();
3226
3227        /* Select a new task to run using either the generic C or port
3228        optimised asm code. */
3229        taskSELECT_HIGHEST_PRIORITY_TASK();
3230
3231
3232        #if( INCLUDE_xTaskCreateTimed == 1 )
3233        {
3234            BaseType_t xHigherPriorityTaskWoken = pdFALSE;
3235
3236            prvStartOverflowTimer( pxCurrentTCB, xIsSwitchContextFromISR, &xHigherPriorityTaskWoken );
3237
3238            if(xHigherPriorityTaskWoken != pdFALSE)
3239            {
3240                /* Select a new task to run using either the generic C or port
3241                optimised asm code.
3242                Highest priority should be the timer daemon. */
3243                taskSELECT_HIGHEST_PRIORITY_TASK();
3244            }
3245        }
3246        #endif /* INCLUDE_xTaskCreateTimed == 1 */
```

Figure 4.1: Starting of the overflow timer from the file task.c

The overrun timer is, in the beginning, just a TickType_t variable xOverrunTicks in the task's control block. It is incremented every tick timed task is running. When it is over the maximal value it start a software timer with a timeout of 1 tick. Timer daemon triggers the overrrun callback one tick later. Code for incrementing (Figure 4.2) is called from the function xTaskIncrementTick which itself is called from the tick interrupt. Overrun timer (of period 1 tick) is started from the context switch function, with code shown in Figure 4.3.

```
5432    /*-----------------------------------------------------------*/
5433
5434    #if INCLUDE_xTaskCreateTimed == 1
5435        BaseType_t prvIncrementOverrunTick(void)
5436        {
5437            BaseType_t xReturn = pdFALSE;
5438            if( pxCurrentTCB->xOverrunTimer != NULL )
5439            {
5440                pxCurrentTCB->xOverrunTicks++;
5441                if(pxCurrentTCB->xOverrunTicks >= ( pxCurrentTCB->xOverrunTicksMax - 1 ) )
5442                {
5443                    xReturn = pdTRUE;
5444                }
5445            }
5446            return xReturn;
5447        }
5448    #endif
5449
```

Figure 4.2: Incrementing of the overrun timer from the file task.c

```
3210        #if( INCLUDE_xTaskCreateTimed == 1 )
3211        {
3212            if( pxCurrentTCB->xOverrunTimer != NULL )
3213            {
3214                if(pxCurrentTCB->xOverrunTicks >= ( pxCurrentTCB->xOverrunTicksMax - 1 ) )
3215                {
3216                    /* Starts the timer with the period of 1. */
3217                    xTimerResetFromISR( pxCurrentTCB->xOverrunTimer, NULL );
3218                    pxCurrentTCB->xOverrunTicks = 0;
3219                }
3220            }
3221        }
3222        #endif /* INCLUDE_xTaskCreateTimed == 1 */
3223
3224        /* Check for stack overflow, if configured. */
3225        taskCHECK_FOR_STACK_OVERFLOW();
3226
3227        /* Select a new task to run using either the generic C or port
3228        optimised asm code. */
3229        taskSELECT_HIGHEST_PRIORITY_TASK();
```

Figure 4.3: Starting of the overrun timer from the file task.c

Next three figures showcase how the timed tasks work. Figure 4.4 shows when is overrun timer triggered. Similarly, Figure 4.5 showcases how the overflow timer works. Finally, Figure 4.6 shows how reseting the timed task suppresses the timeout of overflow and overrun timers.

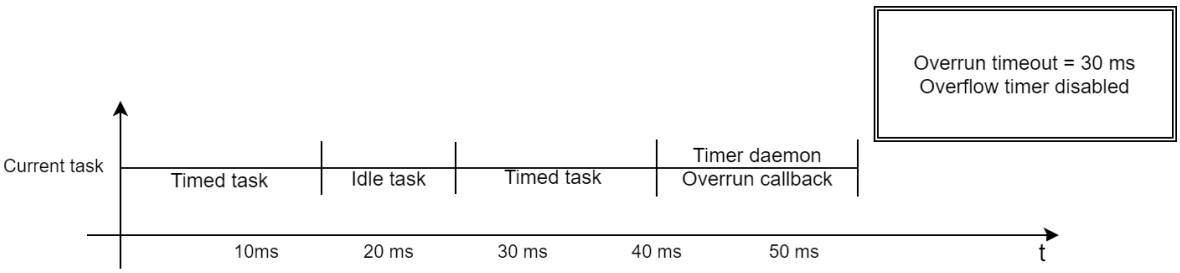

Figure 4.4: Timed task with overrun timeout of 30 ms

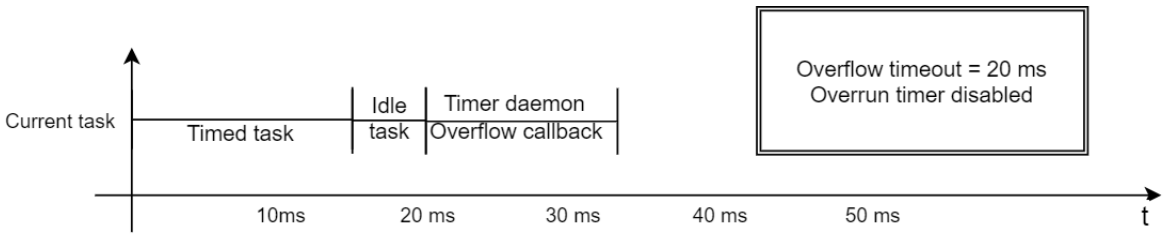

Figure 4.5: Timed task with overflow timeout of 20 ms

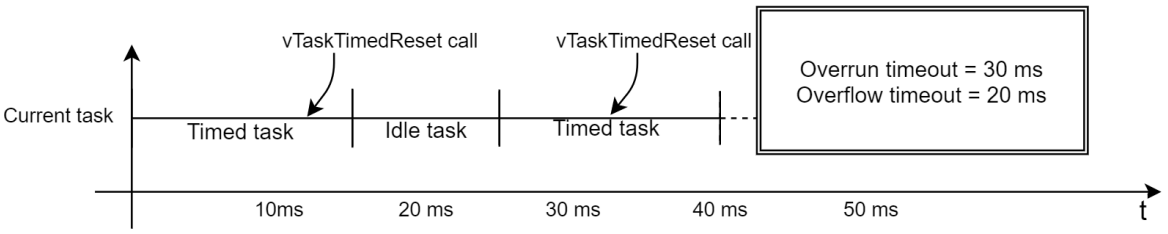

Figure 4.6: Timed task with both timers that resets in time

vTaskDelete function is changed so that deleting tasks also deletes their timers.

4.1.3. Limitiations

Static create of the function is not available.

Timer callback functions are called by the timer daemon and its priority determines when the callback will be called. It is recommended that timer deamon has the highest priority.

4.2. Replicated tasks

4.2.1. Introduction

Redundancy is a common term with safety hardware, but the redundancy can be achieved with the software. As is demonstrated with the replicated tasks. As the name suggests, when replicated tasks are created they make more parallel instances and they compare outputs of each other to assure no errors happened.

Hardware redundancy has the advantage of detecting the fault as early as possible at the cost of increased hardware. On the other hand, software redundancy is useful when the system cost is the restriction ss no additional hardware is needed.
Two types of replicated tasks are implemented:

- 2oo2[1] configuration or without recovery
- 2oo3 configuration or with recovery

Recovery of 2oo3 configuration can be achieved with voting logic. Voting logic can determine which two tasks have the same output and make it a valid one. Same is not possible with 2oo2 voting logic.

[1]MooN is read as M out of N. It shows how many valid outputs have to be present for valid operation e.g. 1oo2 means 1 valid output out of 2 have to be present for a valid operation

4.2.2. Architecture

Replicated tasks have an ability to detect errors using at least two tasks performing identical operations. Tasks are independently processed by the processor. Output variables from tasks are compared in real time. In case of discrepancy in the output variables, an error callback is called where user can process the error.

Figure 4.7 shows how replicated task with recovery works. It shows that all instances wait on the barrier. When all tasks have arrived their compare values are compared. In case of an missmatch, the callback given on task creation is called.

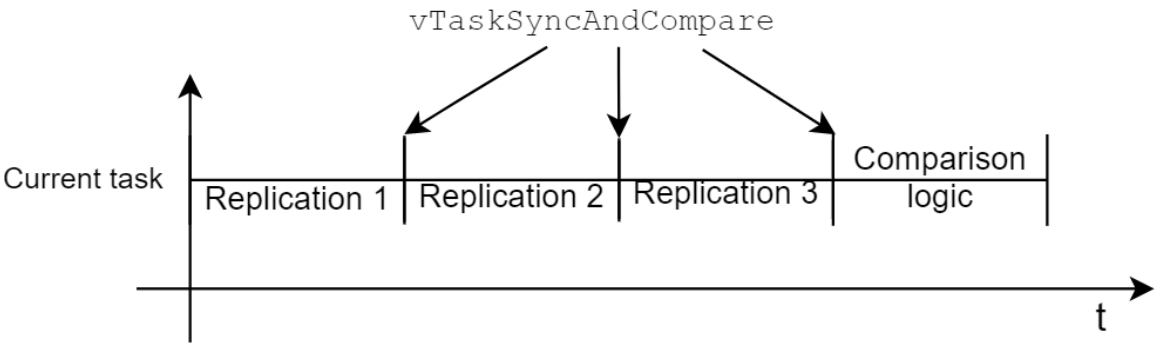

Figure 4.7: Replicated task with redundancy

Comparison logic is not a new task for itself. It is done within one of the tasks. Whichever arrives last. vTaskDelete function was modified so that when one of the replicated sub-tasks delete is requested all linked will be deleted. Tasks are linked over the TCBs[2].

4.2.3. Limitiations

Static create of the timed tasks is not available.

4.3. Command reference

Timed tasks:

- xTaskCreateTimed - Creates a timed task
- vTaskTimedReset - Resets the timer of timed task
- xTimerGetTaskHandle - Gets the corresponding timed task handle from the timer handle

Replicated tasks:

[2]TCB - Task control block

- xTaskCreateReplicated - Creates a replicated task
- xTaskSetCompareValue - Sets a compare value for the calling task
- vTaskSyncAndCompare - Syncronizes the replicated tasks and compares compare values

General added functions:

- eTaskGetType - Get the type of the task
- xTimerPause - Pauses the timer
- xTimerPauseFromISR - Pauses the timer from interrupt service routine
- xTimerResume - Resumes the timer
- xTimerResumeFromISR - Resumes the timer from interrupt service routine
- xTimerIsTimerActiveFromISR - Checks if timer is active from interrupt service routine

4.3.1. xTaskCreateTimed - Creates a timed task.

```
1  BaseType_t xTaskCreateTimed( TaskFunction_t pxTaskCode,
2                       const char * const pcName,
3                       const configSTACK_DEPTH_TYPE usStackDepth,
4                       void * const pvParameters,
5                       UBaseType_t uxPriority,
6                       TaskHandle_t * const pxCreatedTask,
7                       TickType_t xOverrunTime,
8                       WorstTimeTimerCb_t pxOverrunTimerCb,
9                       TickType_t xOverflowTime,
10                      WorstTimeTimerCb_t pxOverflowTimerCb )
```

Create a new timed task and add it to the list of tasks that are ready to run.

Overrun timer is synchronous with the task and its counter is incremented only when timed task is in running state. Overrun callback is called from timer daemon. When timed task overruns it sends a signal to the timer daemon and when callback is called is dependent on daemon's priority. If overrun timer is not used send 0 for xOverrunTime or NULL for the callback.

Overflow timer is asynchronous with the task and its counter is incremented

every tick regardless of the state. Callback is called from timer daemon and its punctuality is dependent on timer daemon's priority. If overflow timer is not used send 0 for xOverflowTime or NULL for the callback.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using xTaskCreateTimed() then both blocks of memory are automatically dynamically allocated inside the xTaskCreate() function. (see http://www.freertos.org/a00111.html). Static version of the function is not implemented.

Input paramters:
 - pvTaskCode - Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).

- pcName - A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN - default is 16.

- usStackDepth -The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage.

- pvParameters - Pointer that will be used as the parameter for the task being created.

- uxPriority - The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE_BIT).

- pvCreatedTask - Used to pass back a handle by which the created task can be referenced.

- xOverrunTime - Runtime of the task after which callback will be called.

- pxOverrunTimerCb - Pointer to the function that will be called if task

runs longer than xOverrunTime without reseting the timed task. Overrun timer is synchronous with the task and its tick is only incremented when timed task is in running state.

- xOverflowTime - Asynchronous timer time. After xOverflowTime pxOverflowTimerCb will be called.

- pxOverflowTimerCb - Pointer to the function that will be called after xOverflowTime. Overflow timer is asynchronous from the task and its value is incremented every tick.

Returns pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h
Example usage:

```
 // Task to be created.
 void vTaskTimedCode( void * pvParameters )
 {
     for( ;; )
     {
         // Task code goes here.

         // Reset the timer.
         vTaskTimedReset(NULL);
     }
 }

// Function to be called if timer overflows.
void vTaskOverflowCallback ( WorstTimeTimerHandle_t xTimer )
{
    // Timeout callback code.

    // Maybe task deletion is needed. Calling vTaskDelete
    ↪  automatically deletes
    // the timer too. Do NOT delete the timer directly. That will
    ↪  cause
    // undefined behavior when deleting the task.
    vTaskDelete( xTimerGetTaskHandle( xTimer ) );
```

```
22 }
23 // Function to be called if timer overflows.
24 void vTaskOverrunCallback ( WorstTimeTimerHandle_t xTimer )
25 {
26
27     // Timeout callback code.
28
29     // Maybe task deletion is needed. Calling vTaskDelete
       ↪ automatically deletes
30     // the timer too. Do NOT delete the timer directly. That will
       ↪ cause
31     // undefined behavior when deleting the task.
32     vTaskDelete( xTimerGetTaskHandle( xTimer ) );
33 }
34
35  // Function that creates a task.
36  void vOtherFunction( void )
37  {
38  static uint8_t ucParameterToPass;
39  TaskHandle_t xHandle = NULL;
40
41      // Create the task, storing the handle.  Note that the passed
       ↪ parameter ucParameterToPass
42      // must exist for the lifetime of the task, so in this case is
       ↪ declared static.  If it was just an
43      // an automatic stack variable it might no longer exist, or at
       ↪ least have been corrupted, by the time
44      // the new task attempts to access it.
45      xTaskCreate( vTaskCode,
46                   "NAME",
47                   STACK_SIZE,
48                   &ucParameterToPass,
49                   tskIDLE_PRIORITY,
50                   &xHandle,
51                   pdMS_TO_TICKS(1 * 1000),
52                   vTaskOverrunCallback,
53                   pdMS_TO_TICKS(2 * 1000),
```

```
                    vTaskOverflowCallback );
    configASSERT( xHandle );

    // Use the handle to delete the task.
    if( xHandle != NULL )
    {
        vTaskDelete( xHandle );
    }
}
```

4.3.2. vTaskTimedReset - Resets the timer of timed task.

```
void vTaskTimedReset( TaskHandle_t pxTaskHandle )
```

Reset the timer of the timed task.

- Warning - Shall only be used for timed tasks.

Input parameters:

- pxTaskHandle - Handle of the task whose timer shall be reset.
Passing a NULL handle results in reseting the timer of the calling task.

Example usage:

```
void vTimedTask( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        vTaskTimedReset(NULL);
    }
}
```

4.3.3. xTimerGetTaskHandle - Gets the corresponding timed task handle from the timer handle.

```
TaskHandle_t xTimerGetTaskHandle( const TimerHandle_t xTimer )
```

Returns the timed task handle assigned to the timer. Task handle is an union with timer ID and that is why they are mutually exclusive.

Task handle is assigned to the timer when creating the timed task.

WARNING: Setting the timer ID also sets the task handle. Changing the timer ID can lead to undefined behavior.

Input parameters:

- xTimer - The timer being queried.

Example usage:

- See xTaskCreateTimed

4.3.4. xTaskCreateReplicated - Creates a replicated task.

```
BaseType_t xTaskCreateReplicated( TaskFunction_t pxTaskCode,
                                  const char * const pcName,
                                  const configSTACK_DEPTH_TYPE
                                  ↪ usStackDepth,
                                  void * const pvParameters,
                                  UBaseType_t uxPriority,
                                  TaskHandle_t * const pxCreatedTask,
                                  uint8_t ucReplicatedType,
                                  RedundantValueErrorCb_t
                                  ↪ pxRedundantValueErrorCb )
```

Create a new replicated task and add it to the list of tasks that are ready to run. Replicated task is used to achieve redundancy of the software at the expense of slower execution. Task executes slower because it is replicated two or three times. Depending on the type chosen. On every call to vTaskSyncAndCompare task is suspended until every replicated task arrives to the same point. When every task is in the synchronization function comparison is done. If any of the comparison results differ callback function pxRedundantValueErrorCb is called. In the callback function user can access the compare values and choose whether to delete all the tasks.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using xTaskCreateReplicated() then both blocks of memory are automatically dynamically allocated inside the xTaskCreateReplicated() function. (see http://www.freertos.org/a00111.html). Static version of this function is not implemented.

Input parameters:
- pvTaskCode - Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).

- pcName - A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN - default is 16.

- usStackDepth - The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage.

- pvParameters - Pointer that will be used as the parameter for the task being created.

- uxPriority - The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE_BIT).

- pvCreatedTask - Used to pass back a handle by which the created task can be referenced.

- ucReplicatedType - Valid values: taskREPLICATED_NO_RECOVERY and taskREPLICATED_RECOVERY. No recovery is faster as it created only two instances, but recovery is not possible. Recovery creates three identical tasks. Recovery is possible with 2 out of 3 logic.

- pxRedundantValueErrorCb - Function to be called when compare values do not match. Return value determines whether calling redundant task will be deleted.

Returns pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

Example usage:

```
// Task to be created.
void vTaskCode( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        vTaskSyncAndCompare(&xCompareValue);
    }
}

// NOTE: This function is called from the redundant task and not daemon.
uint8_t ucCompareErrorCb (CompareValue_t * pxCompareValues, uint8_t ucLen)
{
    // Iterate through compare values.
    for(uint8_t iii = 0; iii < ucLen; i++)
    {
        pxCompareValue[iii]
        .
        .
        .
    }

    return pdTRUE; // Signaling to delete the redundant task.
}

```

```
27 // Function that creates a task.
28 void vOtherFunction( void )
29 {
30 static uint8_t ucParameterToPass;
31 TaskHandle_t xHandle = NULL;
32
33     // Create the task, storing the handle.  Note that the passed
       ↪  parameter ucParameterToPass
34     // must exist for the lifetime of the task, so in this case is
       ↪  declared static.  If it was just an
35     // an automatic stack variable it might no longer exist, or at
       ↪  least have been corrupted, by the time
36     // the new task attempts to access it.
37     xTaskCreateReplicated( vTaskCode, "NAME", STACK_SIZE,
       ↪  &ucParameterToPass, tskIDLE_PRIORITY, &xHandle,
       ↪  taskREPLICATED_RECOVERY, ucCompareErrorCb );
38     configASSERT( xHandle );
39
40     // Use the handle to delete the task.
41     if( xHandle != NULL )
42     {
43         vTaskDelete( xHandle );
44     }
45 }
```

4.3.5. xTaskSetCompareValue - Sets a compare value for the calling task.

```
1 void xTaskSetCompareValue( CompareValue_t xNewCompareValue )
```

Sets the compare value. Compare value is used with replicated tasks. They are used in vTaskSyncAndCompare function for figuring if there is a difference between the tied task executions.
Input parameters:
- xNewCompareValue - New compare value to set.

4.3.6. vTaskSyncAndCompare - Syncronizes the replicated tasks and compares compare values.

```
void vTaskSyncAndCompare( const CompareValue_t * const
↪ pxNewCompareValue )
```

Waits until every replicated task is finished. When every task is finished function compares the compare values and if there is a mismatch it calls the predefined callback.

- Warning - Shall only be used for replicated tasks.

Input parameters:

- pxNewCompareValue - Pointer of the compare value to be copied from. If NULL is passed in, previous compare value is used.

Example usage:

```
void vReplicatedTask( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        vTaskSyncAndCompare(&xCompareValue);
    }
}
```

4.3.7. eTaskGetType - Get the type of the task.

```
eTaskType eTaskGetType( TaskHandle_t pxTaskHandle )
```

Get the type of the task.

Input parameters:

- pxTaskHandle - Handle of the task to be queried. Passing a NULL handle results in getting the type of calling task.

4.3.8. xTimerPause - Pauses the timer.

```
1 BaseType_t xTimerPause( TimerHandle_t xTimer, TickType_t xTicksToWait )
```

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerPause() pauses a timer. If timer was not running before it is ignored. Pausing remembers how many ticks until the deadline are needed and on next xTimerResume() timer will trigger only after the ticks set by pause.

Pausing assures timer is in stopped state.

- xTimer - The handle of the timer being paused.

- TicksToWait - Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the stop command to be successfully sent to the timer command queue, should the queue already be full when xTimerPause() was called. xTicksToWait is ignored if xTimerPause() is called before the scheduler is started.

Returns pdFAIL if the pause command could not be sent to timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue.
When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

4.3.9. xTimerPauseFromISR - Pauses the timer from interrupt service routine.

```
1  BaseType_t xTimerPauseFromISR( TimerHandle_t xTimer,
2                                 BaseType_t *pxHigherPriorityTaskWoken
                                  ↪  );
```

A version of xTimerPause() that can be called from an interrupt service routine.

- xTimer - The handle of the timer being paused.

- pxHigherPriorityTaskWoken - The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerPauseFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerPauseFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerPauseFromISR() function. If xTimerPauseFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Returns pdFAIL if the pause command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

4.3.10. xTimerResume - Resumes the timer.

```
1 BaseType_t xTimerResume( TimerHandle_t xTimer, TickType_t xTicksToWait
  ↪ )
```

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerResume() resumes a timer. If timer was not running before it acts as xTimerStart. If timer saw stopped prior to the call with xTimerPause than it

places a deadline in daemon task from the time timer left of and not the full period.

Resuming assures timer is in running state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the
timer will get called 'n' ticks after xTimerStart() was called, where 'n' is the time left from when last pause was called.

- xTimer - The handle of the timer being resumed.

- xTicksToWait - Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the resume command to be successfully sent to the timer command queue, should the queue already be full when xTimerResume() was called. xTicksToWait is ignored if xTimerResume() is called before the scheduler is started.

pdFAIL will be returned if the resume command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

4.3.11. xTimerResumeFromISR - Resumes the timer from interrupt service routine.

```
1 BaseType_t xTimerResumeFromISR( TimerHandle_t xTimer,
2                                 BaseType_t *pxHigherPriorityTaskWoken
                                  ↪ )
```

A version of xTimerResume() that can be called from an interrupt service routine.

- xTimer - The handle of the timer being resumed.

- pxHigherPriorityTaskWoken - The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the

timer command queue. Calling xTimerPauseFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerPauseFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerPauseFromISR() function. If xTimerPauseFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

pdFAIL will be returned if the resume command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

4.3.12. xTimerIsTimerActiveFromISR - Checks if timer is active from interrupt service routine.

```
1 BaseType_t xTimerIsTimerActiveFromISR( TimerHandle_t xTimer );
```

A version of xTimerIsTimerActive() that can be called from an interrupt service routine.

- xTimer - The handle of the timer that is to be checked.

pdFAIL will be returned if the reset command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerResetFromISR() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

5. Secure bootloader

5.1. What is a bootloader?

Bootloaders are usually the first pieces of code that run, they run just before the user's application e.g. an operating system. They are used to manage the memory. It is highly processor and board specific. The term "bootloader" is a shortened form of the words "bootstrap loader". The term stems from the fact that the boot manager is the key component in starting up the computer, so it can be likened to the support of a bootstrap when putting a boot on.[14]

5.2. Developed bootloader overview

Developed bootloader can be controlled using a command shell communication over UART. The bootloader has an ablility to load new application over UART. In addition, a number of memory management functions are added. When updating the application bootloader accepts three types: binary (.bin), Intel hex (.hex) or Morotola S-record (.srec). Transmitted new application can additionally be checksummed with SHA256 or cyclic redundancy check (CRC32).

The default STM32F407 microcontroller's bootloader doesn't allow the aforementioned functionality and that is the main motivation for writing code for this platform. [18] First version of the bootloader is developed for STM32F407-Discovery board. Bootloader code is situated in the first three sectors of microcontrollers memory, as seen in Table 5.1. Fourth section is used as persistent memory (not loaded on the code startup) for communication between bootloader and user's application. More about application boot record in section 5.5.

Bootlader is written according to the BARR:C-2018 C coding standard to minimize defects in code. [4]

File structure of the bootloader source code for STM32F407 is as follows:

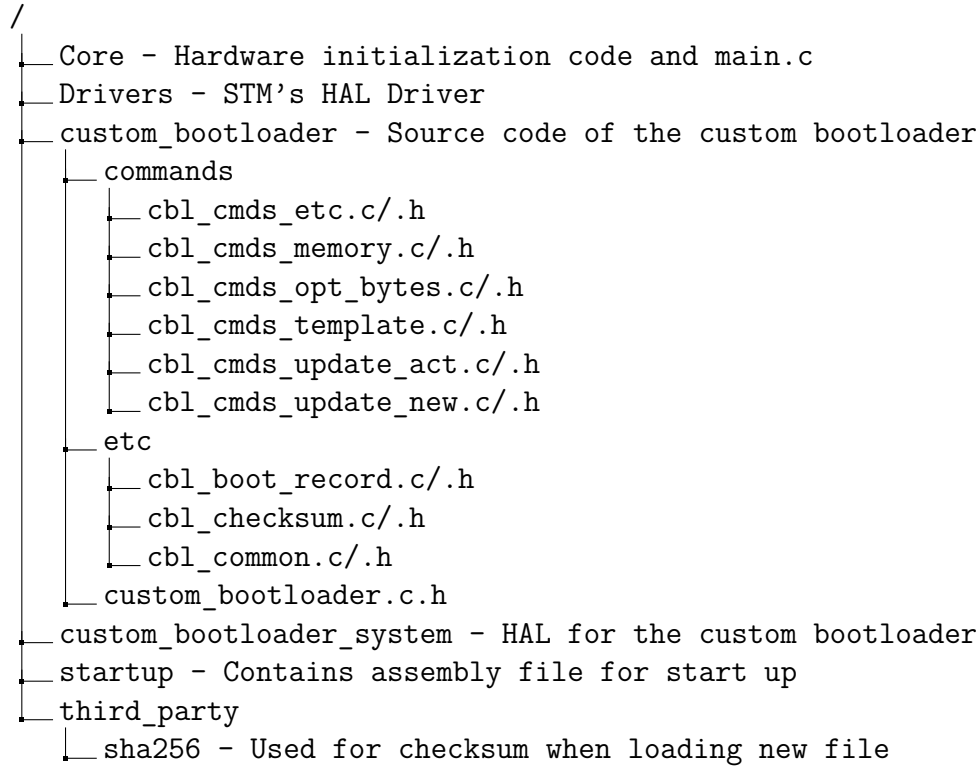

Figure 5.1: Bootloader file structure for STM32F4007 microcontroller

5.3. The bootloader's architecture

The bootloader architecture is simple. On entry, the bootloader checks if blue button on the discovery board is pressed, if it is pressed bootloader is skipped and user's application starts. Bootloader starts otherwise. On bootloader start, it checks if user's application update is needed and updates it if needed. Next step is going into system state machine.

Bootloader has 3 states: Operation, error and exit. Operation state flow is shown in Figure 5.2. Operation state waits for incomming commands and processes them, error state constructs and sends error message back to the user. Exit state is called right before exiting, it is used to deconstruct data from the bootloader.

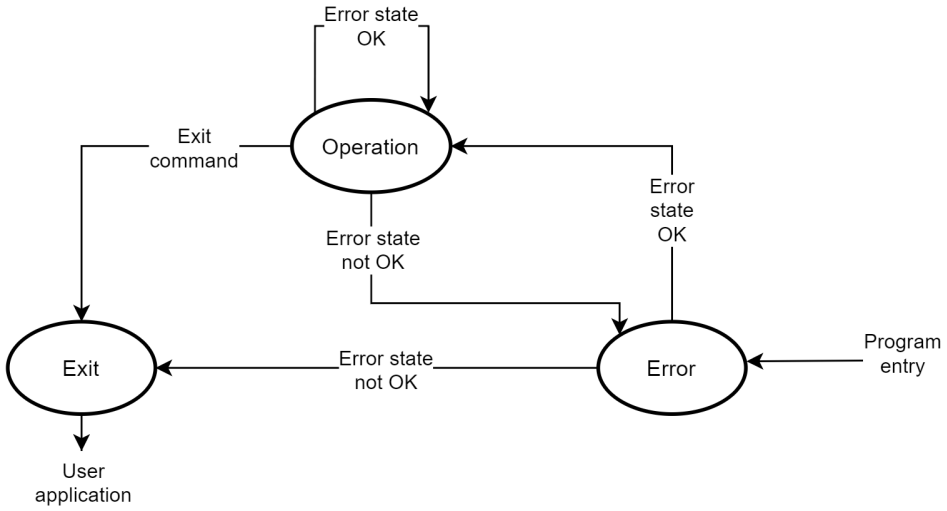

Figure 5.2: State machine of the bootloader

```
> get-write-pro

ERROR: Invalid command

>
```

Figure 5.3: Example of an error from error state

5.4. Flash memory organization

When using the bootloader the flash module is organized a shown in Table 5.1.

Block	Used by	Name	Block base addresses	Size
Main memory	Bootloader	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
		Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
		Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Boot record	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Current application	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
		Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
		Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
		Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
	New application	Sector 8	0x0808 0000 - 0x0809 FFFF	128 Kbytes
		Sector 9	0x080A 0000 - 0x080B FFFF	128 Kbytes
		Sector 10	0x080C 0000 - 0x080D FFFF	128 Kbytes
		Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbytes
System memory			0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area			0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes			0x1FFF C000 - 0x1FFF C00F	16 bytes

Table 5.1: STM32F407 flash memory organization

5.5. Application boot record

Aplication boot record is used to store meta data about the current user's application and new user's application. Meta data consists of:

- Checksum used for transmission,
- Application type used while transmitting,
- Length of application during transmission.

Boot record is also used to signalize that update of the application is needed to the bootloader. Flag is set when new application is successfully transmitted.

Listing 5.1 and Listing 5.2 show modifications added to the linker file needed to add the boot record. Address 0x800C000 is the starting address of sector 3 of the flash memory.

```
/* Specify the memory areas */
MEMORY
{
RAM (xrw)     : ORIGIN = 0x20000000, LENGTH = 128K
CCMRAM (rw)     : ORIGIN = 0x10000000, LENGTH = 64K
/* Allow bootloader only first 3 sectors */
```

```
FLASH (rx)     : ORIGIN = 0x8000000, LENGTH = 48K
/* Allow sector 3 for app boot record */
SEC3 (rx)             : ORIGIN = 0x800C000, LENGTH = 16K
}
```

Listing 5.1: Memory areas from the linker file.

```
 /* Application boot record */
 .appbr 0x800C000 (NOLOAD):
 {
   . = ALIGN(4);
   _sappbr = .;
   *(.appbr)
   *(.appbr*)

   . = ALIGN(4);
   _eappbr = .;
 } >SEC3
```

Listing 5.2: Application boot record from the linker file.

5.6. User application modifications

On the start of every STM32F407 program is a vector table. Vector table contains numerous interrupt and exception vectors. List of all vectors is avaliable in [18, p. 372]. On start up the program calls the vector on the address 4, the name of that vector is fittingly Reset handler. But before calling the reset handler main stack pointer(MSP) is set from the address 0.

Because the program expects the main stack pointer and reset handler vector to be on the start of the program vector offset register(VTOR) is available. Vector offset register is simply added onto the flash memory base address to allow multiple programs in the same flash memory. Perfect for writing a bootloader!

To sum up, before bootloader jumps to the user application it must set the MSP to the one of the user's application then it jumps to the application's reset handler. Vector offset register can be set be the bootloader or in the user's application, former is chosen in this project.

5.7. Command reference

Important notices:

- Every execute of a command must end with \r \n
- Commands are case insensitive
- On error bootloader returns "ERROR:<Explanation of error>"
- Optional parameters are surrounded with [] e.g. [example]

List of all commands:

- version - Gets a version of the bootloader
- help - Makes life easier
- reset - Resets the microcontroller
- cid - Gets chip identification number
- get-rdp-level - Gets read protection [18, p. 93]
- jump-to - Jumps to a requested address
- flash-erase - Erases flash memory
- flash-write - Writes to flash memory
- mem-read - Read bytes from memory
- update-act - Updates active application from new application memory area
- update-new - Updates new application
- en-write-prot - Enables write protection per sector
- dis-write-prot - Disables write protection per sector
- get-write-prot - Returns bit array of sector write protection
- exit - Exits the bootloader and starts the user application

5.7.1. version - Gets a version of the bootloader.

Parameters:

```
 - None
Execute command:

   > version

Response:
```

```
    v1.0
```

5.7.2. help - Makes life easier.

```
Parameters:

 -  None

Execute command:

    > help

Response:

    <List of all available commands and examples>
```

5.7.3. reset - Resets the microcontroller.

```
Parameters:

 - None

Execute command:

    > reset

Response:

    OK
```

5.7.4. cid - Gets chip identification number.

```
Parameters:

 - None

Execute command:
```

```
    > cid

Response:

    0x413
```

5.7.5. get-rdp-level - Gets read protection [18, p. 93]

```
Parameters:

  - None

Execute command:

    > get-rdp-level

Response:

    level 0
```

5.7.6. jump-to - Jumps to a requested address.

```
Parameters:

- addr - Address to jump to in hex format (e.g. 0x12345678), 0x can be
    omitted

Execute command:

    > jump-to addr=0x87654321

Response:

    OK
```

5.7.7. flash-erase - Erases flash memory.

```
Parameters:
```

- type - Defines type of flash erase. "mass" erases all sectors, "sector" erases only selected sectors

- sector - First sector to erase. Bootloader is on sectors 0, 1 and 2. Not needed with mass erase

- count - Number of sectors to erase. Not needed with mass erase

Execute command:

```
> flash-erase sector=3 type=sector count=4
```

Response:

```
OK
```

5.7.8. flash-write - Writes to flash memory.

Parameters:

- start - Starting address in hex format (e.g. 0x12345678), 0x can be omitted

- count - Number of bytes to write, without checksum. Chunk size: 5120

- [cksum] - Defines the checksum to use. If not present no checksum is assumed. WARNING: Even if checksum is wrong data will be written into flash memory!

  - "sha256" - Gives best protection (32 bytes), slowest, uses software implementation

  - "crc32" - Medium protection (4 bytes), fast, uses hardware implementation.

  - "no" - No protection, fastest

Note:

When using crc-32 checksum sent data has to be divisible by 4

Execute command:

> flash-write start=0x87654321 count=64 cksum=crc32

Response:

chunks:1

chunk:0|length:64|address:0x87654321

ready

Send bytes:

<64 bytes>

Response:

chunk OK

checksum|length:4

ready

Send checksum:

<4 bytes>

Response:

OK

5.7.9. mem-read - Read bytes from memory.

Parameters:

- start - Starting address in hex format (e.g. 0x12345678), 0x can be

```
    omitted

- count - Number of bytes to read

Execute command:

    > mem-read start=0x87654321 count=3

Response:

    <3 bytes starting from the address 0x87654321>

Note:
- Entering invalid read address crashes the program and reboot is required.
```

5.7.10. update-act - Updates active application from new application memory area.

```
Parameters:
- [force] - Forces update even if not needed

   - "true" - Force the update

   - "false" - Don't force the update


Execute command:

    > update-act force=true

Response:

    No update needed for user application
    Updating user application
    OK
```

5.7.11. update-new - Updates new application.

```
Parameters:
```

- count - Number of bytes to write, without checksum. Chunk size: 5120

- type - Type of application coding

    - "bin" - Binary format (.bin)

    - "hex" - Intel hex format (.hex)

    - "srec" - Motorola S-record format (.srec)

- [cksum] - Defines the checksum to use. If not present no checksum is assumed. WARNING: Even if checksum is wrong data will be written into flash memory!

    - "sha256" - Gives best protection (32 bytes), slowest, uses software implementation

    - "crc32" - Medium protection (4 bytes), fast, uses hardware implementation.

    - "no" - No protection, fastest

Execute command:

```
> update-new count=4 type=bin cksum=sha256
```

Response:

```
chunks:1

chunk:0|length:4|address:0x08080000

ready
```

Send bytes:

```
<4 bytes>
```

```
Response:

    chunk OK

    checksum|length:32

    ready

Send checksum:

     <32 bytes>

Response:

    OK
```

5.7.12. en-write-prot - Enables write protection per sector.

```
Parameters:

- mask - Mask in hex form for sectors where LSB corresponds to sector 0

Execute command:

    > en-write-prot mask=0xFF0

Response:

    OK
```

5.7.13. dis-write-prot - Disables write protection per sector.

```
Parameters:

- mask - Mask in hex form for sectors where LSB corresponds to sector 0

Execute command:

    > dis-write-prot mask=0xFF0
```

```
Response:

    OK
```

5.7.14. get-write-prot - Returns bit array of sector write protection.

Parameters:

```
- None

Execute command:

    > get-write-prot

Response:

    0b100000000010
```

5.7.15. exit - Exits the bootloader and starts the user application.

```
Parameters:

- None

Execute command:

    > exit

Response:

    Exiting
```

6. Developed software demonstration

6.1. Overview

This chapter showcases how the FreeRTOS modifications and the developed secure bootloader work together. Firstly, the boot-up of the bootloader is shown. After the bootloader, the example from the FreeRTOS source code demonstrating the replicated tasks is demonstrated. Next up, microcontroller is restarted and update of the user application is requested. The process of loading a new application is showcased. Loaded new application is actually an example of the timed tasks. Finally, after the load, bootloader jumps to an example of the timed tasks from the FreeRTOS modifications.

6.2. Bootloader boot-up

On boot-up, bootloader firstly checks if the blue button is pressed on the STM32F407 DISC1 development board. If the button is not pressed, bootloader checks whether the update of the user application is needed i.e. it checks the application boot record where the meta data of the current and new applications are stored.

Figure 6.1 shows the feedback to the user on the boot-up. Firstly the preable is printed, followed up by the `No update needed for user application` which is printed after the application boot record is checked. Finally, `>` is printed and bootloader is awaiting the user's command and the blue LED is turned on.

```
1  ********************************************
2  Custom bootloader for STM32F4 Discovery board
3  ********************************************
4  ********************************************
5                      v1.1
6  ********************************************
7                Master's thesis
8                   Dino Saric
9              University of Zagreb
10                     2020
11 ********************************************
12           If confused type "help"
13 ********************************************
14 No update needed for user application
15
16 OK
17
18 >
```

Figure 6.1: Bootloader boot-up when no update is needed

Figure 6.2 showcases the version command of the bootloader.

```
1  > version
2  v1.1
3
4  OK
5
6  >
```

Figure 6.2: Bootloader version command

6.3. Replicated tasks example

This example is run by setting the `EXAMPLE_REPLICATED` in `FreeRTOS_demonstration/example/Inc/example.h` on line 20 to 1 and disabling timed and default example. Source code that will make the demonstration easier to follow is located in

`FreeRTOS_modification/example/src/example_replicated.c`. In the demonstration four replicated tasks are present. All of them with different configuration. Table 6.1 lists all tasks in the demonstration.

Task name	Replicated recovery (number of sub-tasks)	Always same compare values
Repl recov ok	true (3)	true
Repl recov fail	true (3)	false
Repl non-recov ok	false (2)	true
Repl non-recov fail	false (2)	false

Table 6.1: All replicated tasks in demonstration

Figure 6.3 shows the first 11 seconds of the replicated tasks example. Number before the `':'` is time elapsed from the FreeRTOS start. First, task `Repl recov ok` arrives to the `vTaskSyncAndCompare`. There are three print-outs because three sub-tasks exist and are going over the same code. When the last sub-tasks arrives to the `vTaskSyncAndCompare` all tasks are compared. Their value matches and no callback is called, rather the tasks are unblocked and enter the 5 second delay.

Line 6, which is printed after 2 seconds of execution, shows `Repl non-recov ok` task which enters the sync and compare function. After the comparison both sub tasks resume, as their compare values match. On line 8, aforementioned `Repl recov ok` again arrives to the compare point where it is again compared. Its compare value matches and the task is resumed.

After 6 seconds, task `Repl recov fail` arrives to the compare point. After all three tasks have arrived they are compared. Comparison shows that not all compare values match. Two values are 0, but one is a 1. Next line shows that the software successfully deduced with voting logic that the correct result is 0 (line 16).

Right after the task `Repl recov fail` is done, task `Repl non-recov fail` arrives at the compare point. Like the last task before, the compare values don't match (line 20). The difference is that no result could be deduced this time as there are only two sub-tasks.

After the last task is finished, task `Repl non-recov ok` again comes to the compare function and the logic is repeated. All tasks are periodic with period of 5.

```
1  Jumping to user application :)
2  FreeRTOS Modification.
3  0:Task "Repl recov ok" is waiting for comparison.
4  4:Task "Repl recov ok" is waiting for comparison.
5  9:Task "Repl recov ok" is waiting for comparison.
6  2001:Task "Repl non-recov ok" is waiting for comparison.
7  2006:Task "Repl non-recov ok" is waiting for comparison.
8  5014:Task "Repl recov ok" is waiting for comparison.
9  5019:Task "Repl recov ok" is waiting for comparison.
10 5024:Task "Repl recov ok" is waiting for comparison.
11 6027:Task "Repl recov fail" is waiting for comparison.
12 6032:Task "Repl recov fail" is waiting for comparison.
13 6037:Task "Repl recov fail" is waiting for comparison.
14 6042:Task "Repl recov fail" compare failed.
15 6047:Compare values are: 0 0 1
16 6050:Deduced result is 0.
17 6052:Task "Repl non-recov fail" is waiting for comparison.
18 6058:Task "Repl non-recov fail" is waiting for comparison.
19 6063:Task "Repl non-recov fail" compare failed.
20 6068:Compare values are: 0 1
21 6071:Result could not be deduced.
22 7011:Task "Repl non-recov ok" is waiting for comparison.
23 7016:Task "Repl non-recov ok" is waiting for comparison.
24 10029:Task "Repl recov ok" is waiting for comparison.
25 10034:Task "Repl recov ok" is waiting for comparison.
26 10039:Task "Repl recov ok" is waiting for comparison.
27 11053:Task "Repl recov fail" is waiting for comparison.
28 11058:Task "Repl recov fail" is waiting for comparison.
29 11063:Task "Repl recov fail" is waiting for comparison.
30 11069:Task "Repl recov fail" compare failed.
31 11073:Compare values are: 1 1 0
32 11076:Deduced result is 1.
```

Figure 6.3: Replicated tasks example output

6.4. New application loading

Figure 6.4 demonstrates the application flow when the user requests a user application update. To update the application user enters the function `update-new`, more about the function at its reference subsection 5.7.11. Line 1 shows an erroneous call of the function, bootloader returns a descriptive error to the user. After the second call, all the required parameters are entered correctly. Device erases the chunks that are needed by the application, turning the blue LED signalizing that flash is being deleted.

After erasing, bootloader returns the total number of chunks the file will need to be split into. Following, on line 9, chunk 0 information is printed. Finally, `ready` is printed, signalizing the bootloader is ready to receive 5120 bytes. While writing the data sent by the user, the blue LED is turned on emphasizing that the flash memory is being written into. After the data is transmitted bootloader returns `chunk OK` to signalize that the sent bytes were successfully written into flash. The procedure is repeated for the rest of the chunks.

When all the chunks have been written into flash, if the checksum was specified in the function call, it is awaited by the bootloader. This process is seen on the line 12. When the 32 bytes have been entered, inputted and calculated checksums are compared and `OK` is printed.

After the successful checksum, the microcontroller is automatically restarted. On the restart, it works out that the user application needs an update and it updates the user application by copying the bytes from new application (as described in Table 5.1) into the current application flash area.

```
> update-new count=757720 type=hex cksum=sha256
ERROR: New app is too long. Aborting
> update-new count=75772 type=hex cksum=sha256
chunks:15
chunk:0|length:5120|address:0x08080000
ready
chunk OK
<chunks 1 - 13>
chunk:14|length:4092|address:0x08091800
ready
chunk OK
checksum|length:32
ready
<32 checksum bytes>
OK
Restarting...
*********************************************
Custom bootloader for STM32F4 Discovery board
*********************************************
*********************************************
                   v1.1
*********************************************
              Master's thesis
                 Dino Saric
           University of Zagreb
                   2020
*********************************************
         If confused type "help"
*********************************************
Update for user application available
Updating user application
OK
>
```

Figure 6.4: New application loading output

6.5. Timed tasks example

After the successful update of the user application, timed task example is loaded into the flash. Timed example demonstrates how overrun and overflow timer work. Overrun timer increments only when the timed task is active. Overflow timer increments asynchronously to the task, i.e. it increments regardless if the timed task is active.

Figure 6.5 shows how the first 30 seconds of the example from `FreeRTOS_modification/example/src/example_timed.c`. The matching output is achieved by setting the `EXAMPLE_TIMED` to 1 on line 19 in `FreeRTOS_demonstration/example/Inc/example.h`. Table 6.2 shows all the tasks in the example.

Task name	Overflow timeout (s)	Overrun timeout (s)	Period (ms)	Priority, higher is more important
Oflow task ok	6	-	5900	3
Oflow task fail	6	-	-	2
Oflow task fail and del	6	-	-	3
Orun task ok	-	5	4900	3
Orun task fail	-	5	-	2
Orun task fail and dele	-	5	-	3

Table 6.2: All timed tasks in demonstration

In Figure 6.5, first task that resets the timers using the function `vTaskTimedReset` is the `Orun task ok`, it resets the timers before the overflow timer triggers so no callback is called. Next task that resets its timers is `Oflow task ok`, it also resets the timers before the overrun timer triggers and the callback is not called. On line 10, overflow timer from task `Oflow task fail and del` overflows and it deletes itself from the callback. Right after, overflow timer from task `Oflow task fail` triggers.

On line 12, first overrun timer is triggered. The callback deletes the task after it has been triggered. This trigger is made possible using the `HAL_Delay()` function inside the task with timeout larger than the overrun timeout. On line 16, the overrun timer of the task `Orun task fail` triggers. Signalizing that the task was in running state for 5 seconds, during which overrun timer was not reset.

```
> exit

OK
Exiting

Jumping to user application :)
FreeRTOS Modification.
4900:Task "Orun task ok" is reseting the timer.
5900:Task "Oflow task ok" is reseting the timer.
6000:Task "Oflow task fail and del" overflowed. Deleting task.
6006:Task "Oflow task fail" overflowed. This happens every 6 s.
8120:Task "Orun task fail and dele" overran. Deleting task.
9808:Task "Orun task ok" is reseting the timer.
11809:Task "Oflow task ok" is reseting the timer.
12000:Task "Oflow task fail" overflowed. This happens every 6 s.
12239:Task "Orun task fail" overran. This happens after 5 s of
    runtime.
14712:Task "Orun task ok" is reseting the timer.
17713:Task "Oflow task ok" is reseting the timer.
18000:Task "Oflow task fail" overflowed. This happens every 6 s.
18258:Task "Orun task fail" overran. This happens after 5 s of
    runtime.
```

Figure 6.5: Timed tasks example output

7. Conclusion

This thesis explores the functional safety practices in embedded systems. Overview of functional safety and related standards is provided to help engineers without functional safety experience. The thesis clarifies how ARM Cortex R improves functional safety principles over ARM Cortex M.

A software is developed to achieve software redundancy in ARM Cortex M. Implemented software includes a bootloader and a modified FreeRTOS operating system. The bootloader has the ability to load new applications and other memory manipulation related operations. The bootloader also supports permanent memory for communication between bootloader and the current application. To aid security for loading of a new application support for HEX and SREC format is provided, along with additional checksum appended to the data.

The modified FreeRTOS operating system provides features of tracking the individual tasks execution time and task replication. Tasks tracking their time have two timers, one for overrun and one for overflow. Overrun timers count only when task is active. Overflow timer always counts regardless of the task's state. Replicated tasks can have two or three redundant tasks, when three tasks are used replicated task is fault tolerant. Demonstratration of the functionality is available through examples in source code. Inner workings of FreeRTOS and the bootloader is provided in this thesis.

Bibliography

[1] ARM. *Cortex-R5 technical reference manual Revision: r1p2*, 2011. URL https://static.docs.arm.com/ddi0460/d/ddi0460d_cortex_r5_r1p2_trm.pdf.

[2] ARM. *Cortex-R5 technical reference manual Revision: r1p2*, 2011. URL https://static.docs.arm.com/ddi0460/d/ddi0460d_cortex_r5_r1p2_trm.pdf.

[3] *ARM Cortex M4 Processor - Technical Reference Manual*. ARM, 2020. URL https://developer.arm.com/documentation/100166/0001/.

[4] Michael Barr. *BARR-C:2018 Embedded C Coding Standard*. Barr Group, 2018. URL https://barrgroup.com/sites/default/files/barr_c_coding_standard_2018.pdf.

[5] Richard Barry. *Mastering the FreeRTOS™ Real Time Kernel*, 2016. URL https://www.freertos.org/wp-content/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf.

[6] IEC International Electrotechnical Comission. Briefing paper: Functional safety essential to overall safety, 2020. URL https://basecamp.iec.ch/download/functional-safety-essential-to-overall-safety/.

[7] IEC International Electrotechnical Comission. Functional safety explained, 2020. URL https://www.iec.ch/functionalsafety/explained/.

[8] exida. Functional safety fundamentals, 2019. URL https://www.youtube.com/watch?v=srwHBvEe9WA.

[9] Stylianos Ganitis. Lockstep analysis for safety critical embedded systems, 2015.

[10] Texas Instruments Deutschland GmbH. Delayed lock-step cpu compare, 2008. URL https://patents.google.com/patent/US20080244305A1/en.

[11] Rich Goyette. *An Analysis and Description od the Inner Workings of the FreeRTOS Kernel*, 2007. URL http://richardgoyette.com/Research/Papers/FreeRTOSPaper.pdf.

[12] Texas Instruments. *Safety Manual for TMS570LS31x and TMS570LS21x Hercules™ ARM®-Based Safety Critical Microcontrollers*, 2015. URL https://www.ti.com/lit/ug/spnu511d/spnu511d.pdf?ts=1598992005375.

[13] Texas Instruments. *Technical reference manual for TMS570LS31x/21x*, 2018. URL https://www.ti.com/lit/ug/spnu499c/spnu499c.pdf?ts=1598992969037.

[14] IONOS. Bootloader: What you need to know about the system boot manager, 2020. URL https://www.ionos.com/digitalguide/server/configuration/what-is-a-bootloader/.

[15] Ivan Pavić. *Lockstep in the context of reliability and fault-tolerance of computer systems*, 2019.

[16] ARM Research. *A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications*, 2016. URL https://ieeexplore.ieee.org/abstract/document/7575387.

[17] Amazon Web Services. Mastering the freertos™ real time kernel, 2020. URL https://www.freertos.org/a00114.html.

[18] *STM32F407 reference manual*. STMicroelectronics, 2019. URL https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced.pdf.

Software and Hardware Architecture for Redundant Embedded Systems

Abstract

Functional safety and functional safety standards overview is given, along with an example of functional safety in embedded systems. Explanation of how ARM Cortex-R improves functional safety over Cortex M is provided. Implemented software for redundancy on Cortex M. The software includes a bootloader and a modified FreeRTOS operating system. The bootloader supports the ability to load new applications and contains a command shell with accompanying functions. The modified FreeRTOS operating system provides features of tracking the individual tasks execution time and task replication. Demonstratration of the functionality is available through examples in source code.

Keywords: Functional safety, IEC 61508,IEC 26262, bootloader, certification, ARM Cortex R, Cortex R, FreeRTOS, Cortex M, kernel, redudancy, lock-step

Programska i sklopovska arhitektura redundantnih ugradbenih računalnih sustava

Sažetak

Opisana je funkcijska sigurnost i dan je pregled odgovarajućih standarda. Dodan je i primjer funkcijske sigurnosti u ugradbenim računalnim sustavima. Opisano je kako ARM Cortex R popravlja funkcijsku sigurnost u odnosu na ARM Cortex M. Implementirana je programska potpora za ostvarivanje redundancije na ARM Cortex M mikrokontroleru. Programska potpora za redundanciju sadržava bootloader i modificirani operacijski sustav FreeRTOS. Bootloader podržava mogućnost učitavanja novih aplikacija i naredbenu ljusku s odgovarajućim funkcijama. Modificirani operacijski sustav FreeRTOS ima mogućnost praćenja vremena izvođenja pojedinih zadataka i sposobnost repliciranja zadataka (engl. task replication). Demonstracija funkcionalnosti je dostupna kroz primjere u izvornom kodu.

Ključne riječi: Funkcijska sigurnost, IEC 61508, IEC 26262, bootloader, certifikacija, ARM, Cortex R, FreeRTOS, Cortex M, jezgra, redundancija