UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS num. 1981

Software and Hardware Architecture for Redundant Embedded Systems

Dino Šarić

Zagreb, August 2020.

Umjesto ove stranice umetnite izvornik Vašeg rada.

Kako biste uklonili ovu stranicu, obrišite naredbu `\izvornik`*.*

Contents

List of Figures

Todo list

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.[4] 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.[4]

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.

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. [5]

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[6]:

– The systematic capability rating,

– The architectural constrains for the element and

– The PFDaverage calculation.
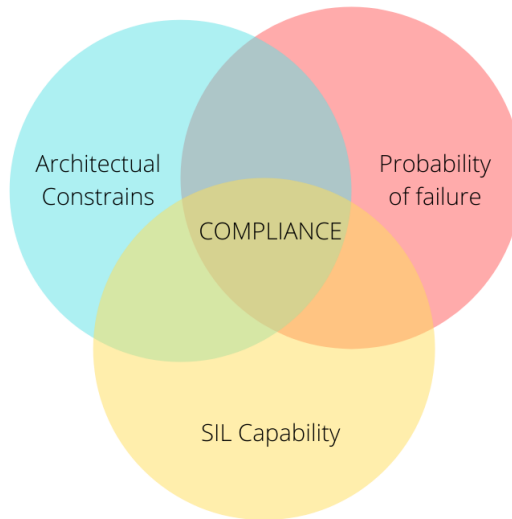
Figure 1.2: SIL requirements

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.

Figure 1.3: ARM Cortex-R5 Processor SIL 3 certificate

1.4. Embedded processors redundancy - lockstep

Consider a system with one processing unit. Such system is obviously non-redundant from the aspect of instruction execution. Lockstep is redundancy mechanism for in-

creasing 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. [10]

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). [10]

1.4.1. Example use case [7]

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 comparison with 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, electrionic 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. Theese 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. [1] 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-R5 processor lockstep

2.3.1. Dual core lockstep

dual code lockstep, mention delayed lockstep CPU compare

2.3.2. Triple core lockstep

dual code lockstep

2.3.3. Sto jos dodaje Cortex R

data cache, instruction cache, atcm, btcm, preformance monitor unit, interrupt
controller connected though porr, more pipeline stages

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. [11]

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. [3]

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. [3]

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. [3]

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. [8] A FreeRTOS task can exist in five states: running, blocked, ready, suspended and deleted. A state diagram is shown in Figure 3.2.
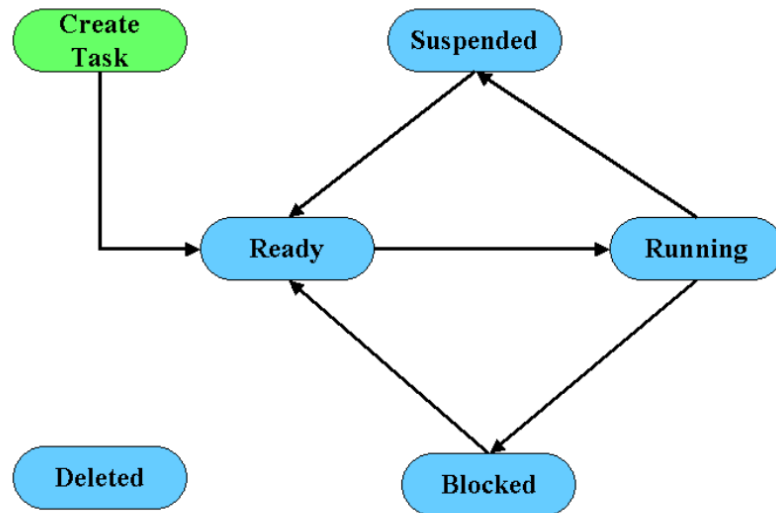
Figure 3.2: FreeRTOS task states[8, p 10]

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.
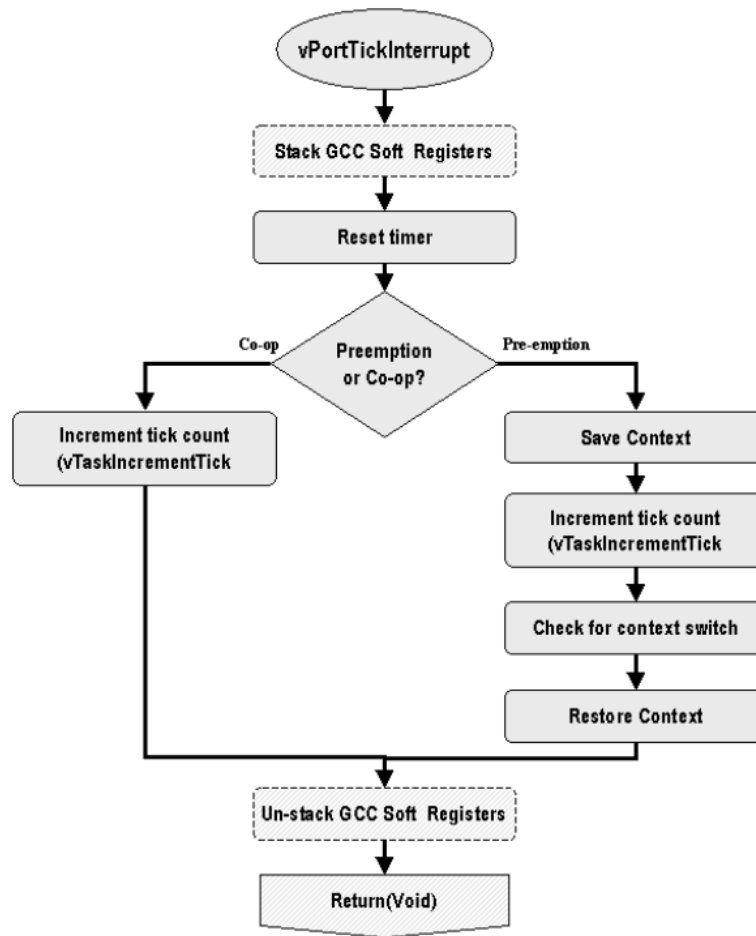
Figure 3.3: Scheduler algorithm[8, p 20]

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.
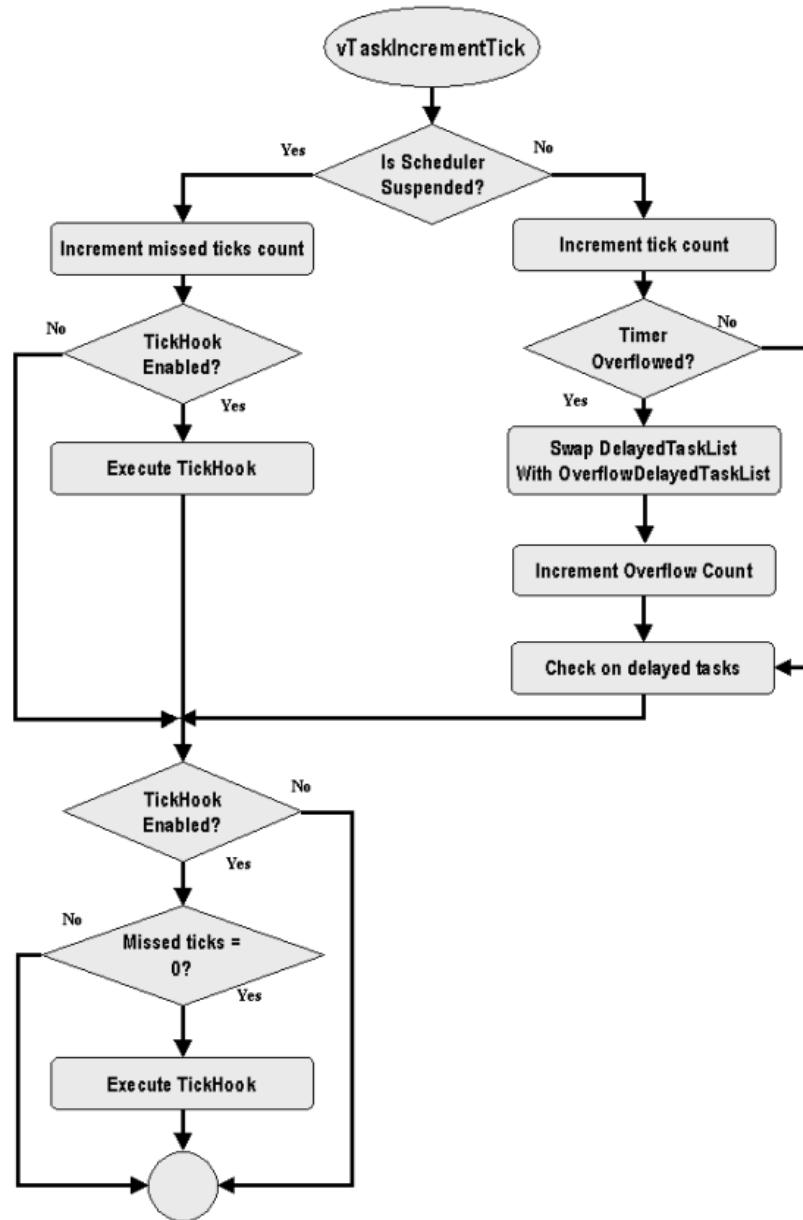
Figure 3.4: vTaskIncrementTick algorithm[8, p 31]

More about the scheduler can be found in [8].

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.
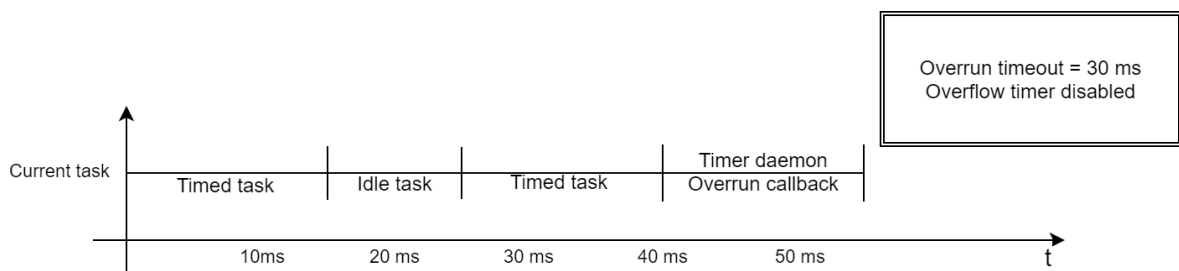
Figure 4.4: Timed task with overrun timeout of 30 ms
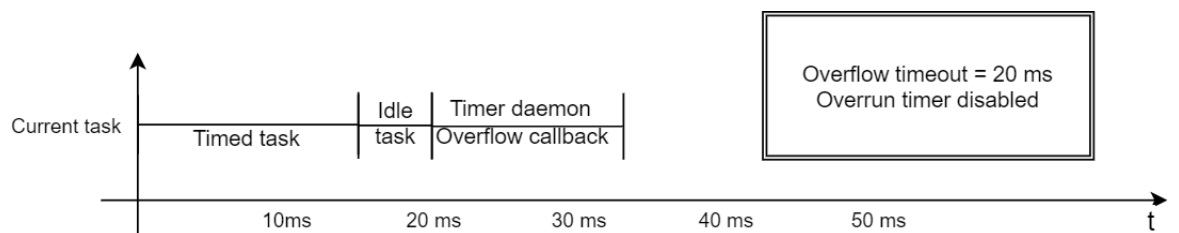
Figure 4.5: Timed task with overflow timeout of 20 ms
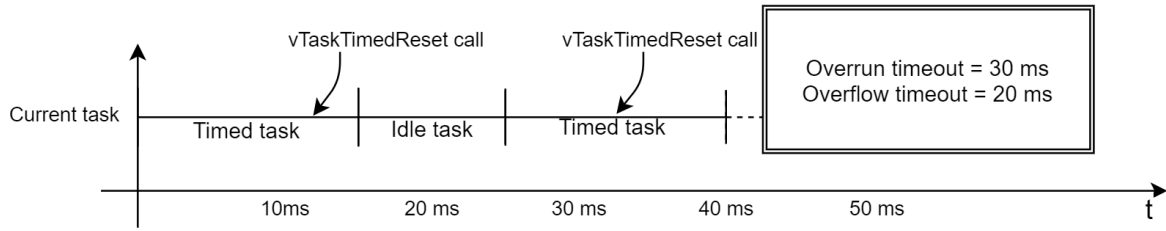
Figure 4.6: Timed task with both timers that resets in time

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.
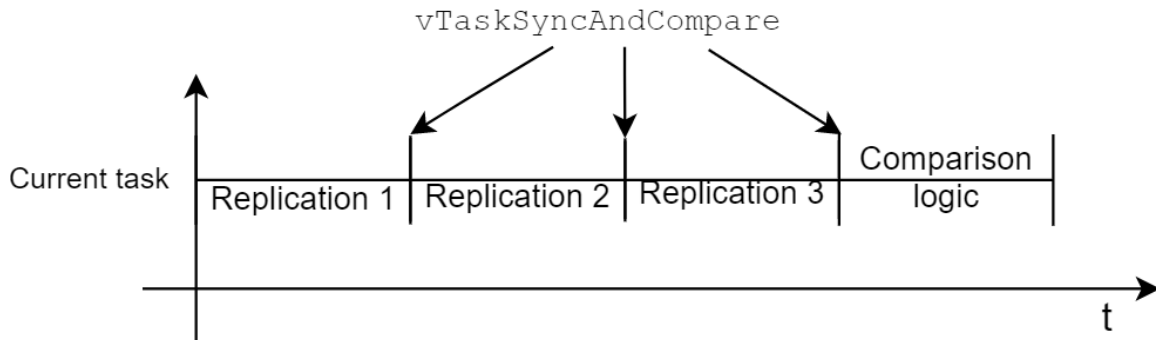
Figure 4.7: Replicated task with redundancy

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.

```
BaseType_t xTaskCreateTimed( TaskFunction_t pxTaskCode,
                    const char * const pcName,
                    const configSTACK_DEPTH_TYPE usStackDepth,
                    void * const pvParameters,
                    UBaseType_t uxPriority,
                    TaskHandle_t * const pxCreatedTask,
                    TickType_t xOverrunTime,
                    WorstTimeTimerCb_t pxOverrunTimerCb,
                    TickType_t xOverflowTime,
                    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 resetting 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:

```
1   // Task to be created.
2   void vTaskTimedCode( void * pvParameters )
3   {
4       for( ;; )
5       {
6           // Task code goes here.
7
8           // Reset the timer.
9           vTaskTimedReset(NULL);
10      }
11  }
12
13  // Function to be called if timer overflows.
14  void vTaskOverflowCallback ( WorstTimeTimerHandle_t xTimer )
15  {
16      // Timeout callback code.
17
18      // Maybe task deletion is needed. Calling vTaskDelete
          ↪   automatically deletes
19      // the timer too. Do NOT delete the timer directly. That will
          ↪   cause
20      // undefined behavior when deleting the task.
21      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),
```

```
54                    vTaskOverflowCallback );
55        configASSERT( xHandle );
56
57        // Use the handle to delete the task.
58        if( xHandle != NULL )
59        {
60            vTaskDelete( xHandle );
61        }
62   }
```

4.3.2. vTaskTimedReset - Resets the timer of timed task.

```
1  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:

```
1  void vTimedTask( void * pvParameters )
2  {
3      for( ;; )
4      {
5          // Task code goes here.
6
7          vTaskTimedReset(NULL);
8      }
9  }
```

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

```
1  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.

```
1  BaseType_t xTaskCreateReplicated( 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                                     uint8_t ucReplicatedType,
8                                     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:

```
1   // Task to be created.
2   void vTaskCode( void * pvParameters )
3   {
4       for( ;; )
5       {
6           // Task code goes here.
7
8           vTaskSyncAndCompare(&xCompareValue);
9       }
10  }
11
12  // NOTE: This function is called from the redundant task and not
    ↪  daemon.
13  uint8_t ucCompareErrorCb (CompareValue_t * pxCompareValues, uint8_t
    ↪  ucLen)
14  {
15      // Iterate through compare values.
16      for(uint8_t iii = 0; iii < ucLen; i++)
17      {
18          pxCompareValue[iii]
19          .
20          .
21          .
22      }
23
24      return pdTRUE; // Signaling to delete the redundant task.
25  }
26
```

```
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.

```
1  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:

```
1  void vReplicatedTask( void * pvParameters )
2  {
3      for( ;; )
4      {
5          // Task code goes here.
6
7          vTaskSyncAndCompare(&xCompareValue);
8      }
9  }
```

4.3.7. eTaskGetType - Get the type of the task.

```
1  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.[9]

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. [12] 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. [2]

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

```
/
├── Core - Hardware initialization code and main.c
├── Drivers - STM's HAL Driver
├── custom_bootloader - Source code of the custom bootloader
│   ├── commands
│   │   ├── cbl_cmds_etc.c/.h
│   │   ├── cbl_cmds_memory.c/.h
│   │   ├── cbl_cmds_opt_bytes.c/.h
│   │   ├── cbl_cmds_template.c/.h
│   │   ├── cbl_cmds_update_act.c/.h
│   │   └── cbl_cmds_update_new.c/.h
│   ├── etc
│   │   ├── cbl_boot_record.c/.h
│   │   ├── cbl_checksum.c/.h
│   │   └── cbl_common.c/.h
│   └── custom_bootloader.c.h
├── custom_bootloader_system - HAL for the custom bootloader
├── startup - Contains assembly file for start up
└── third_party
    └── sha256 - Used for checksum when loading new file
```

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.

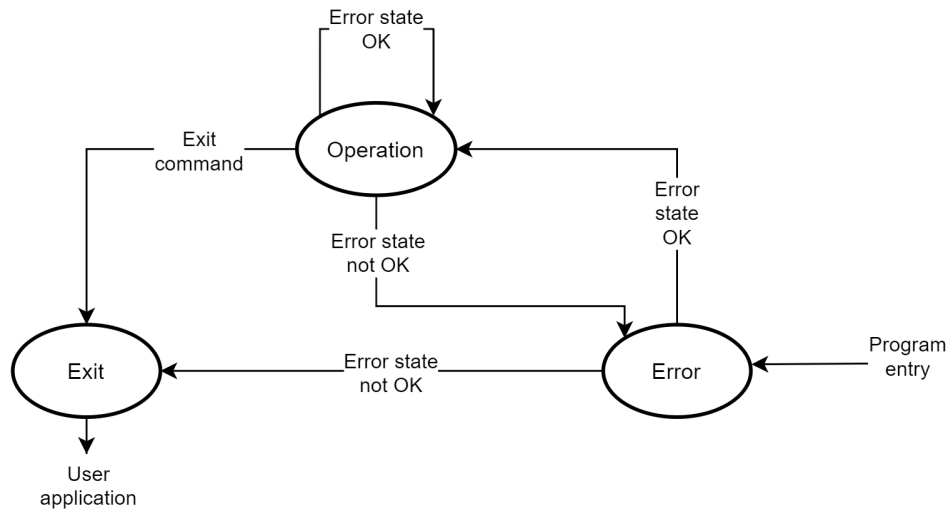| Block	Used by	Name	Block base addresses	Size
		Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Bootloader	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
		Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Boot record	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
		Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Current	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
Main memory	application	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
		Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
	New	Sector 8	0x0808 0000 - 0x0809 FFFF	128 Kbytes
		Sector 9	0x080A 0000 - 0x080B FFFF	128 Kbytes
		Sector 10	0x080C 0000 - 0x080D FFFF	128 Kbytes
	application	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 [12, 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 [12, 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 [12, 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. Settings in [Apendix A](#apend_a)

  - "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. Settings in [Apendix A](#apend_a)

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

[2] 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.

[3] 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.

[4] 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/.

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

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

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

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

[9] 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/.

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

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

[12] *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: Funkocionalna sigurnost, IEC 61508, IEC 26262, bootloader, certifikacija, ARM, Cortex R, FreeRTOS, Cortex M, jezgra, redundancija