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`.

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 all the helpful current and former students from the college forums. Finnally, thanks to the Youtubers that made understanding the college curriculum much easier.

Contents

List of Figures

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.[2] 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.[2]

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 inner workings and what do they add over Cortex M. Chapter 3 gives overview of added safety functions to the FreeRTOS kernel and gives a brief overview of FreeRTOS's inner workings. Chapter 4 explains how the developed secure bootloader functions and its features.

1. Functional safety in embedded systems

Opis functional safety-a, definicija, proces... **TODO**

HW **TODO**

SW **TODO**

2. Cortex R additions over cortex M

TODO uniti koje imaju pojedine cortex r implementacije **TODO** Sto cortex M ima

3. FreeRTOS functional safety additions

TODO Motivacija, zasto 1oo2D i 2oo3D. **TODO** Opis FreeRTOS-a. **TODO** Motivacija za timed tasks, periodicki taskovi Jeleknovic. **TODO** Opis FreeRTOS-a.

4. Secure bootloader

4.1. 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.[1]

4.2. 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.

The default STM32F407 microcontroller's bootloader doesn't allow the aforementioned functionality and that is the main motivation for writing code for this platform. [3] 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 4.1. 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 4.5.

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:

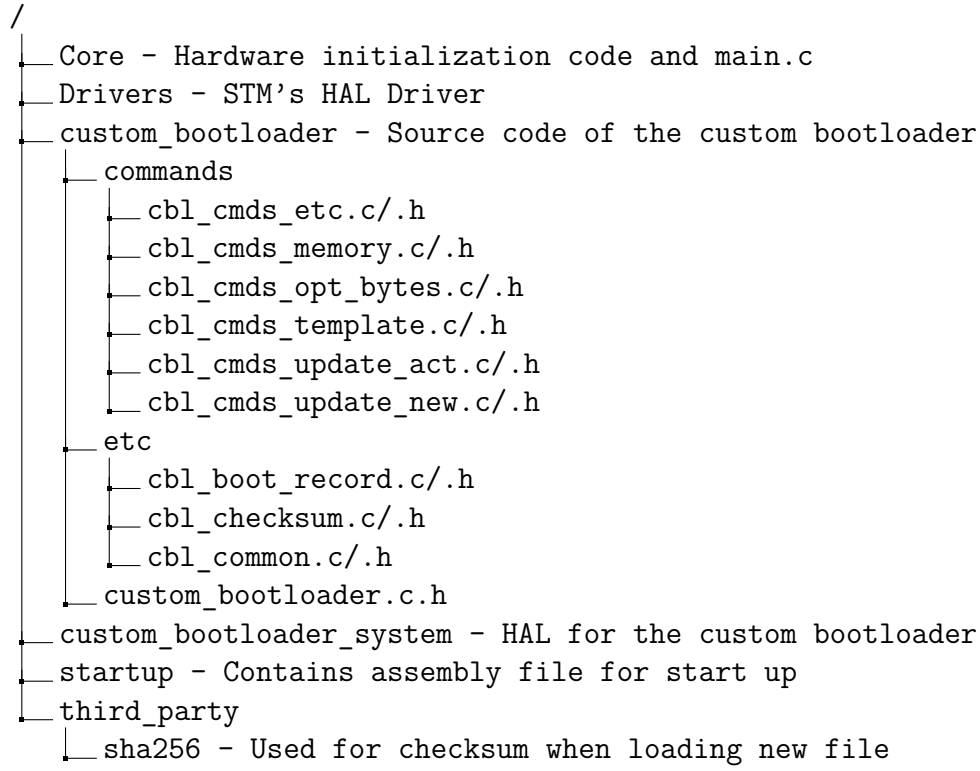

Figure 4.1: Bootloader file structure for STM32F4007 microcontroller

4.3. 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 4.2. 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.

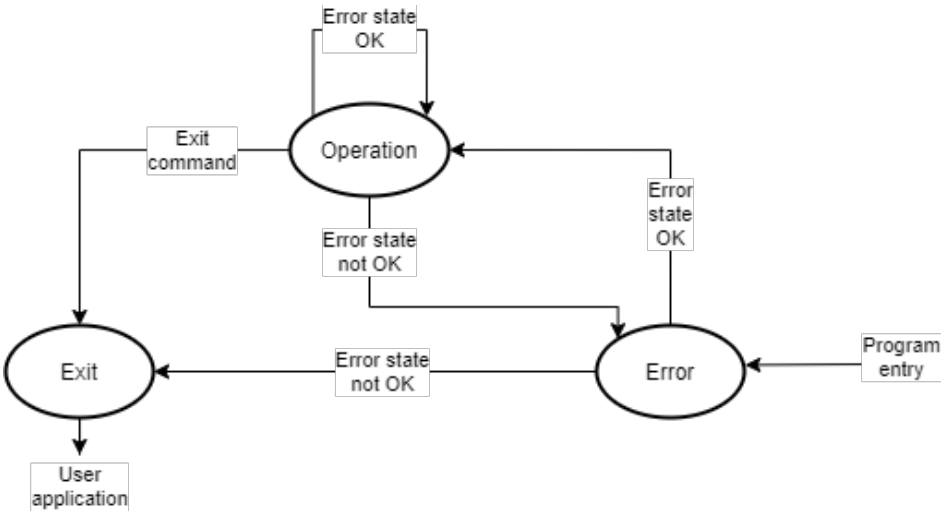

Figure 4.2: State machine of the bootloader

```
> get-write-pro

ERROR: Invalid command

>
```

Figure 4.3: Example of an error from error state

4.4. Flash memory organization

When using the bootloader the flash module is organized a shown in table 4.1.

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 4.1: STM32F407 flash memory organization

4.5. 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.

Listings 4.1 and 4.2 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 4.1: Memory areas from the linker file.

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

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

Listing 4.2: Application boot record from the linker file.

4.6. 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 [3, 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.

4.7. Supported functions

TODO Kako se updatea aplikacija **TODO** Dodati da prihvaca srec i hex

5. Conclusion

In this thesis a functional safety overview was presented.

TODO

Bibliography

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

[2] Briefing paper: Functional safety essential to overall safety. URL https://basecamp.iec.ch/download/functional-safety-essential-to-overall-safety/.

[3] *STM32F407 reference manual*. URL https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f
pdf.

[4] Michael Barr. Barr-c:2018 embedded c coding standard. URL https://barrgroup.com/sites/default/files/barr_c_coding_standard_2018.pdf.

Software and Hardware Architecture for Redundant Embedded Systems

Abstract

Abstract.

Keywords: Keywords.

Programska i sklopovska arhitektura redundantnih ugradbenih računalnih sustava

Sažetak

Sažetak.

Ključne riječi: Ključne riječi, odvojene zarezima.