Self-critical Smart Toaster

Final project report

Embedded Systems Application on CTU Prague

Petr Kučera

May 24, 2024

1 Description of designed embedded device

1.1 Target and motivation

Do you have a regular toaster at home? What if you could turn it into a smart device that prepares warm toast for you at a precise time? So when you go to bed at night, you prepare everything, and in the morning, you have warm toast ready on your plate.

The goal of my project is to create a device that will allow controlling the toaster through a user interface. The device will enable delayed start.

1.2 Base architecture description

The device is based on STM32H747I-DISCO, and I use a relay to control the toaster's power.[1] Users control the device through a touchscreen display. The device run internal timers to count down the time for the toaster to start and how long it should run. To ensure safety, the microcontroller utilize both processors, which mutually check each other in all their activities.

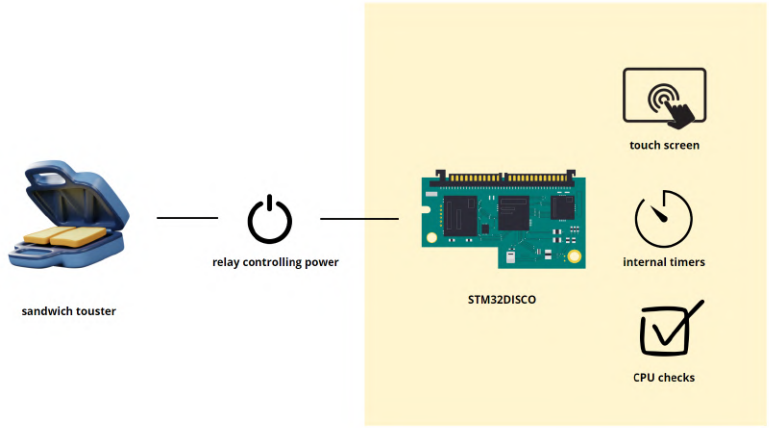

Figure 1: Ilustration schema of project

[1] For example, this one: https://dratek.cz/arduino/7522-rele-ssr-solid-state-ssr-40da-380v.html.

1.2.1 Realization

I have made the following changes to the final project compared to the original specification:

- I am not using hardware timers to determine the runtime; instead, I am using a software-emulated timer. More details in the section 2.
- The cores communicate with each other but only validate the toaster startup, as it is the most potentially life-threatening process.
- For testing, I did not use relays but only utilized built-in LED diodes.
- The final GUI differs from the MockUp designs mainly due to testing during actual operation.

2 Description of the used HW

The MCU controls relay using GPIO output pins and the pins are protected by 470 Ω resistors as ilutrated in the figure 2. The input control voltage of relay is 3-24 VDC and output voltage of relay is 24-240 VAC. For higher loads, it's recommended to add passive or active cooling to the relay. However, in our case, we won't utilize it because we'll assume that the toaster will only be used for short periods at a time.

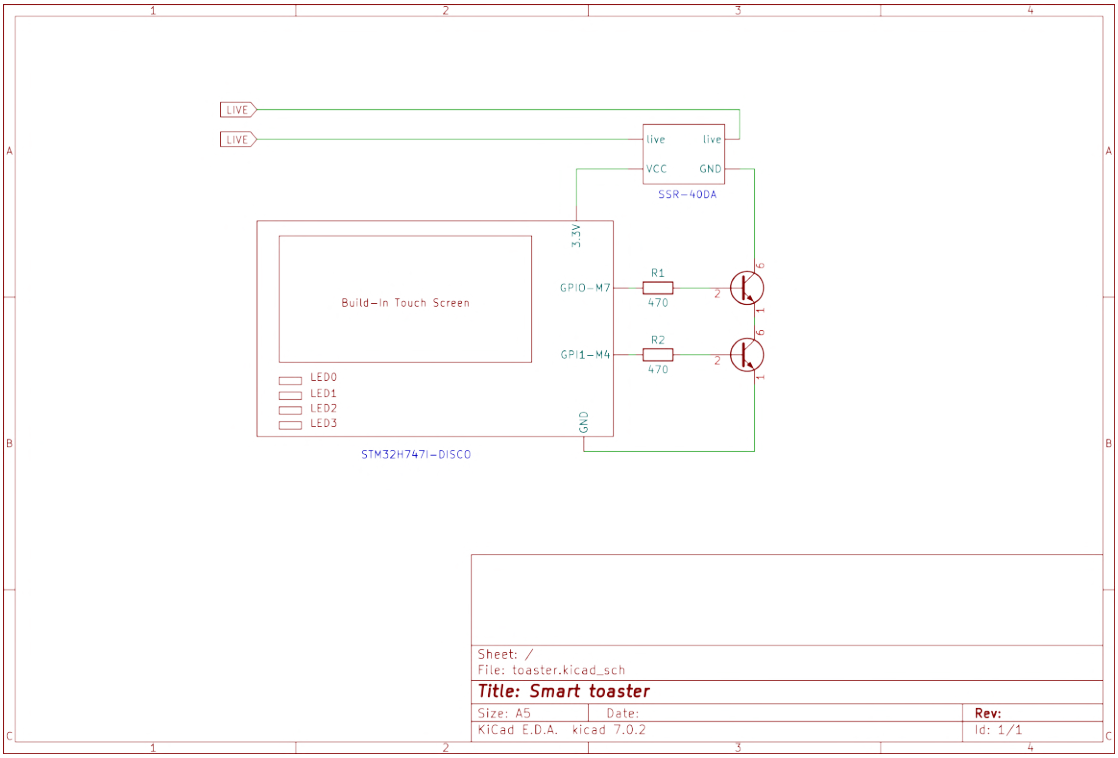

Figure 2: Schema of wiring MCU and Relay

3 Description of the software solution

The software solution is based on an STM template[2] for controlling the LCD display. It uses the HAL and BSD libraries, and the STM driver for the FT6x06[3] to control the touch screen. The code is implemented in C and is available in the repository.[4]

3.1 Startup and booting

CPU1 (Cortex-M7) and CPU2 (Cortex-M4) are booting at once (with respect to configured boot Flash options) System Init, System clock, voltage scaling and L1-Cache configuration[5] are done by CPU1 (Cortex-M7). In the meantime Domain D2 is put in STOP mode (CPU2: Cortex-M4 in deep sleep mode) to save power consumption. When system initialization is finished, CPU1 (Cortex-M7) could release CPU2 (Cortex-M4) when needed by means of HSEM notification or by any D2 Domain wakeup source (SEV, EXTI..).

The above will guarantee that CPU2 (Cortex-M4) code execution starts after system initialization : (system clock config, external memory configuration..).

3.2 Used peripheals

The firmware does not use any peripherals to interact with the environment. It only uses built-in peripherals, which primarily include:

- Built-in LED diodes, these are used for event signaling. Their exact usage is described in section 3.3.2.
- DSI for communication with the display: Communication with the display is done using a single buffer and in command mode.
- I2C and GPIO for communication with the touch screen, for which I use the FT6x06 driver.
- Internal RCC.
- HSEM for initial communication between the cores.

3.3 Firmware structure and flowchart

Upon power-up, the program initializes. We will not discuss this further here as it is described in the chapter 3. The general program flow is described in the flowcharts in the figure 3 and consists of four basic states, referred to as scenes in the implementation.

1. The first scene, **FRONT SCREEN**, displays the initial screen. In this state, the display shows the header, footer, and basic control buttons, which allow the user to either manually start the toaster or go to the delayed start configuration.

[2] Used template is *LCD_DSI_CmdMode_SingleBuffe*: https://github.com/STMicroelectronics/STM32CubeH7/
[3] The FT6x06 driver repository: https://github.com/STMicroelectronics/stm32-ft6x06
[4] The project repository: https://github.com/petrkucerak/safe-critical-toaster
[5] The L1-Cache is in finale product diseabled, it couse problems in multicore communication (it implemented by using a shared memory).

2. The second state is **TURN ON SCENE**, during which the toaster is running. In the background of this scene, a timer counts down the runtime. This value is fixed in the code and serves as a safety measure, ensuring the toast does not burn but is well-toasted. The remaining time is shown in the footer progress bar. This state can be exited using the MANUALLY STOP button.

3. The third state is **TIMER CONFIG SCENE**. This scene is used to configure the TIMER, specifically for delayed start. Using the touch controls, the user can select the delay time. The default delay time is set in the code. From this scene, the TIMER can be started.

4. The final scene is called **WAITING SCENE**. In this state, the controller waits for the set delay to pass, then transitions to the TURNON SCENE. The timer can be skipped with the MANUALLY START button or turned off with the STOP TIMER button.

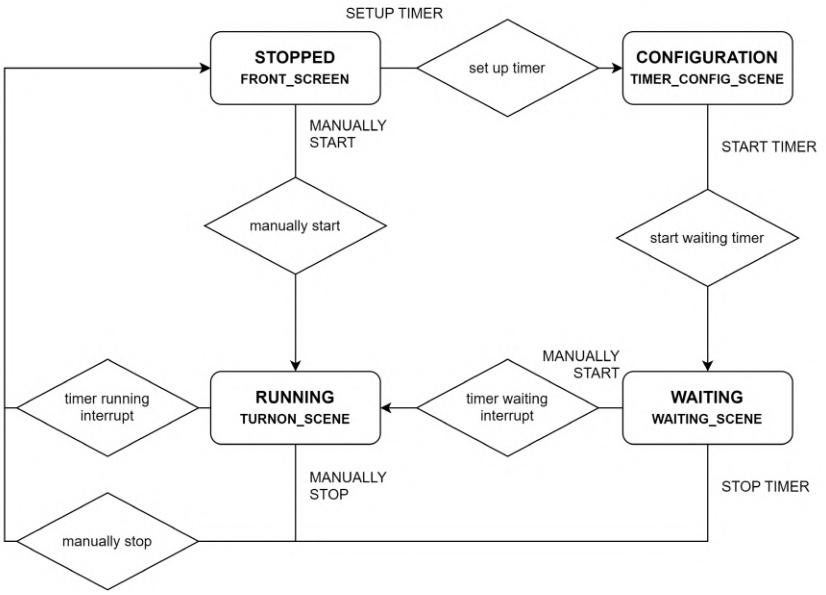

Figure 3: Status diagram

The main part of the program runs on the M7 core without using interrupts, in a loop that performs the following steps:

1. Checks if there has been a touch on the display.

2. If there has been a touch, it calculates the potential areas and possibly modifies the configuration of the control structure.

3. Updates the timer if it is running.

4. Updates and redraws the display.

5. Turns on the toaster and communicates with the M4 core.

As mentioned earlier, the code uses a control structure common to all main control functions. Interrupts are allocated for routines handling peripherals such as the display, I2C, and others.

3.3.1 Software Timer

As previously mentioned, the timer is not hardware-based but uses the HAL function,[6] which returns the number of ticks since startup, or the number of milliseconds since the core started. The counter uses a variable of type unsigned 32-bit integer.[7] Thus, an overflow occurs approximately every 41 days. This problem has not yet been addressed in the device.

3.3.2 Multicore comunnication

The used MCU has 2 cores, M7 and M4. Communication between them occurs using HSEM notifications and shared memory.

HSEM notifications are primarily used during device startup. During runtime, communication utilizes shared memory and atomic variables. It is a relatively simple yet robust communication system. Each core has its own buffer and can access the buffer of the other core, with access controlled by atomic variables. Shared memory is located in RAM at address *0x38000000* with a size of 64K.[8]

3.3.3 LED Signaling

The M4 core uses **LED1** and **LED2**. LED1 flashes regularly at cca 1 second intervals to indicate the core's operation. LED2 simulates the relay activation for the M4 core.

The M7 core is assigned **LED3** and **LED4**. LED3 is used to signal errors in SRAM memory, and LED4 simulates the relay activation for the M7 core.

3.4 Description of all function

All used functions are documented directly in the code, which is available in the project repository.[9]

4 Description of GUI

The user interface is managed by an LCD with dimensions of 480 by 800 pixels and a touchscreen. The interface uses 3 primary colors: yellow for informational or configuration purposes, green for starting or initializing actions, and red for stopping or indicating a stop.

The interface is based on a mockup specification, which can be seen in the figure 4. The colors are in the so-called ARGB code.

4.1 GUI screenshots

The system renders 4 scenes. Detail of this scenes is describes in section 3.2.

- Front scene (figure 5)
- Turn on scene (figure 6)
- Timer config scene (figure 7)
- Timer waiting scene (figure 8)

[6] Mentions HAL function is 'HAL_GetTick()'.
[7] 'uint32_t'
[8] For the memmory definition is nescessary to config linker script.
[9] Project reposiotry: https://github.com/petrkucerak/safe-critical-toaster.

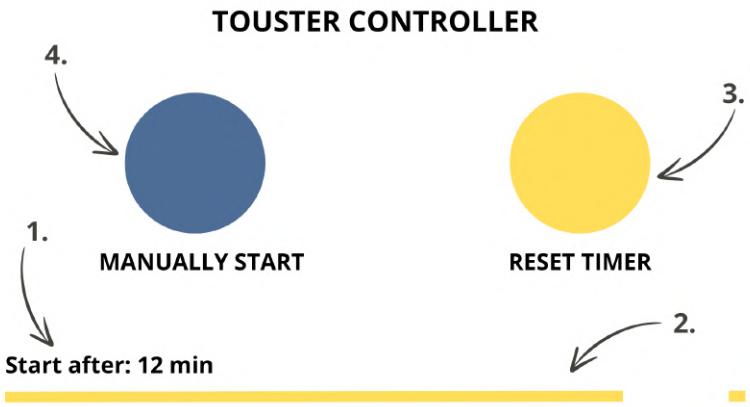

Figure 4: The UI proposal with parts

5 Conclusion and self-reflection

The aim of the project was to implement a solution that would allow for the delayed start of an electrical device, in our case, a toaster. The solution should be controlled via a touchscreen and implement inter-process communication. I succeeded in fulfilling all these points in the solution. However, I realize that the solution is only a concept. To put it into practice, it would be necessary to optimize the used libraries and debug potential errors arising from device usage. The project also does not address all known issues, such as the overflow of the SysTick counter.

If I were to tackle this problem again, I would do certain things differently. For another implementation, I would use a slightly different stack: MXCube for configuration generation, Make for compilation, GCC for compiling, and OpenOCD for uploading. Also, before implementing any driver, I would check if it is already handled by a library. In my case, I started implementing the FT6x06 driver, which, as I later discovered, was already implemented in the BSP library. Initially, I also struggled with importing files in CubeIDE. However, this solution proved logical because it allows the same code to be used for multiple cores, thus preventing data redundancy.

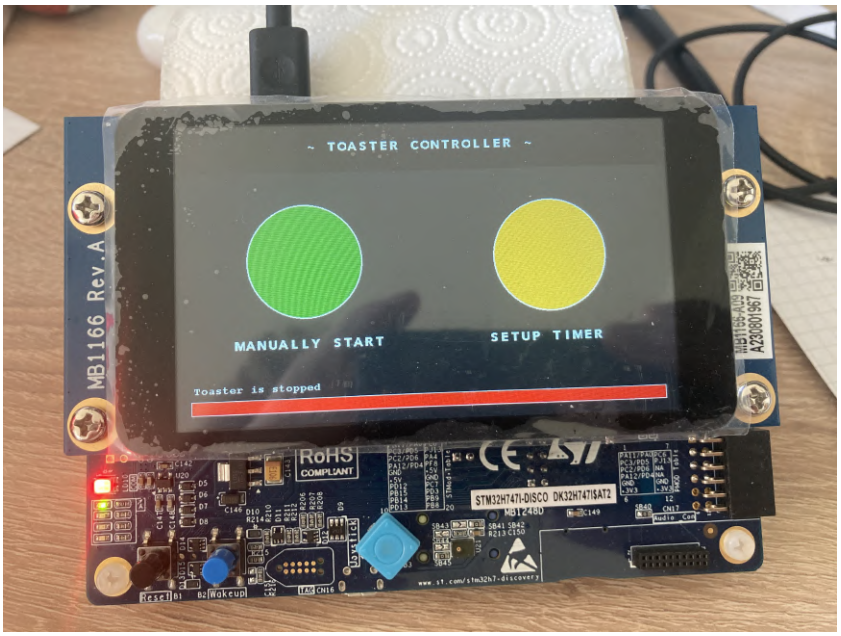

Figure 5: GUI - Front sceen

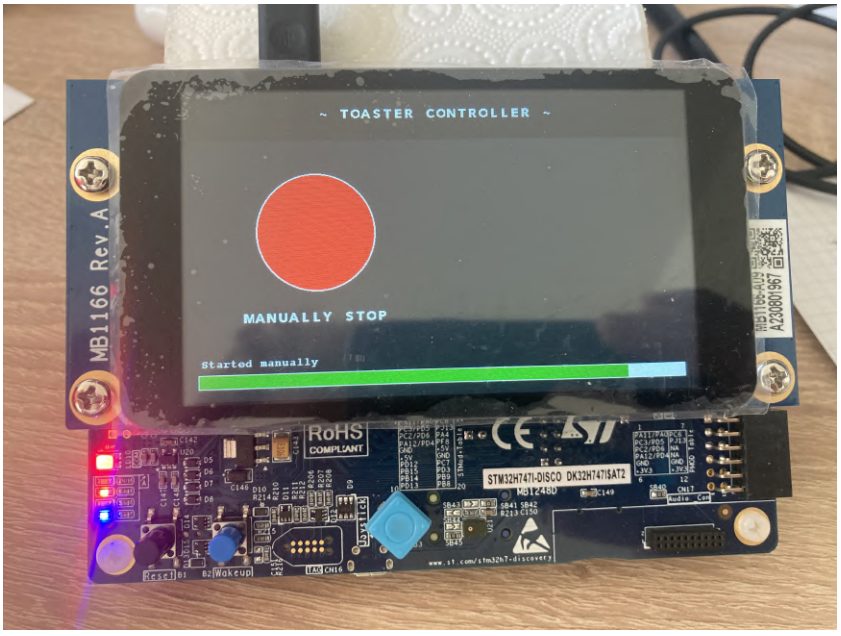

Figure 6: GUI - Turn on screen

Figure 7: GUI - Timer config

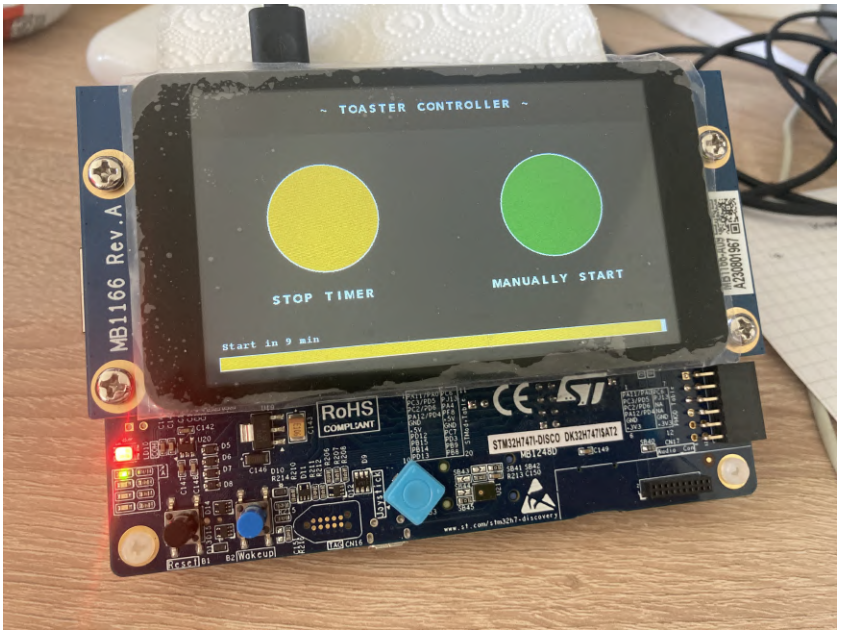

Figure 8: GUI - Timer waiting