Custom Bootloader

Utilizing The STM32 NUCLEO-G071RB Microcontroller

By: Aryan Karnati

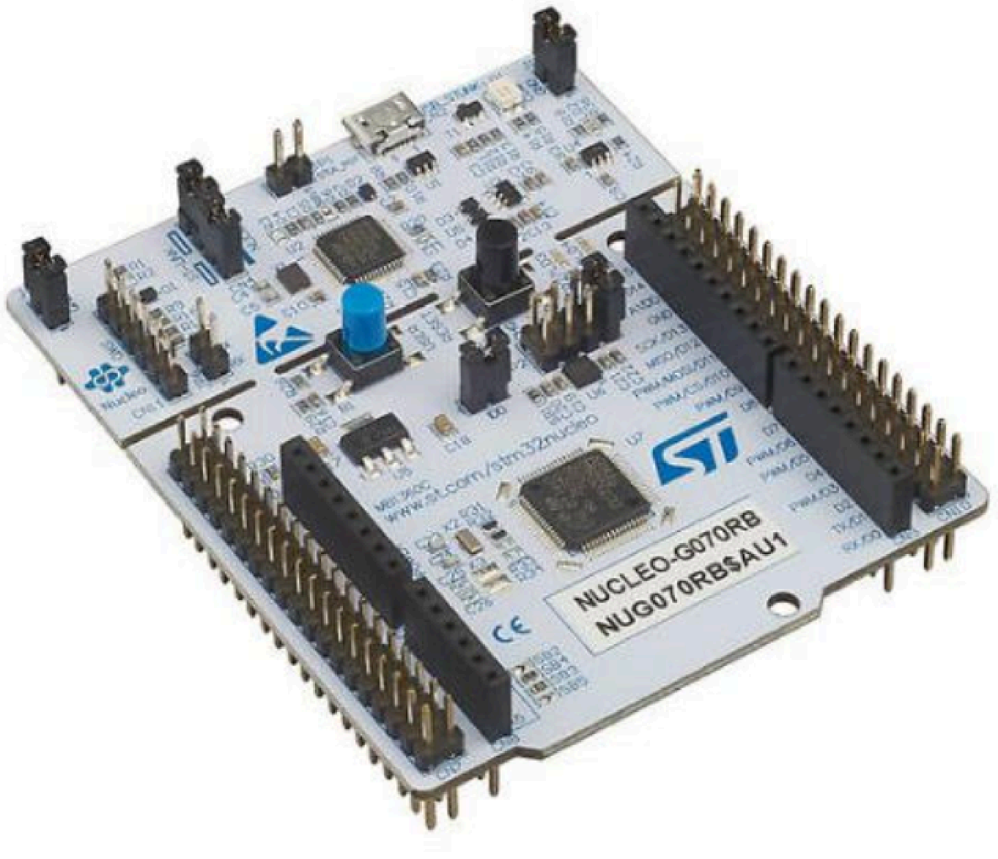

Figure 1: STM32 NUCLEO-G071RB Microcontroller

Table Of Contents

Project Overview

This project demonstrates the development of a custom bootloader and application architecture for the STM32G071RB microcontroller, utilizing STM32CubeIDE. The project highlights key embedded development skills such as memory mapping, linker script customization, firmware modularization, and runtime control handoff. It is designed to reflect industry-relevant practices for building scalable and upgradable embedded systems.

Objectives

The primary objectives of this project are:

- To create a robust bootloader that initializes the system and conditionally jumps to a user application located at a predefined memory address
- To implement a custom linker script that divides the flash memory into distinct regions for the bootloader and the application
- To establish a reliable interface for potential API sharing between the bootloader and the application using linker section attributes
- To design an extendable firmware architecture that can be adapted for Over-The-Air (OTA) or Controller Area Network (CAN)-based upgrades
- To ensure safe and isolated code zones that enable reliable firmware update mechanisms in production environments

Features

- **Custom Linker Script**: Divides flash memory into bootloader and application regions
- **Bootloader-to-Application Handoff**: Controlled jump logic to transfer execution
- **Section-Based API Sharing**: Use of attribute((section())) for potential shared data/functions
- **Extendable Firmware Architecture**: Adaptable for OTA or CAN-based upgrades
- **Safe and Isolated Code Zones**: Ensures reliable firmware update mechanisms

Tools Used

- **Board**: STM32 NUCLEO-G071RB
- **IDE**: STM32CubeIDE
- **Language**: C
- **Debugging**: UART (optional), STM32 HAL Drivers

Bootloader Overview

Introduction

A bootloader is a small program that runs when a microcontroller or microprocessor is powered on or reset. Its primary function is to initialize the hardware and load the main application firmware into memory. Bootloaders are essential for embedded systems as they provide a mechanism for updating firmware without requiring specialized programming hardware.

How Does A Bootloader Work

1. **Power-On or Reset**: When the system is powered on or reset, the microcontroller starts executing code from a predefined memory address, typically the start of the flash memory.
2. **Initialization**: The bootloader initializes the system hardware, including the clock, memory, and peripherals.
3. **Firmware Validation**: The bootloader checks if a valid application firmware is present in a specific memory location. This can involve checking a magic number, a checksum, or a digital signature.
4. **Conditional Jump**: If valid firmware is found, the bootloader sets the stack pointer to the start of the application and jumps to the application's reset vector. If no valid firmware is found, the bootloader may enter a firmware update mode.
5. **Firmware Update Mode**: In this mode, the bootloader waits for new firmware to be received via a communication interface such as UART, USB, or CAN. Once received, the bootloader writes the new firmware to the application memory region and verifies its integrity.

Benefits Of A Bootloader

A bootloader provides a streamlined and efficient method for updating firmware without requiring specialized programming hardware. This is particularly advantageous for embedded systems deployed in the field, where physical access to the device may be limited. By enabling firmware updates over standard communication interfaces such as UART, USB, or CAN, bootloaders reduce maintenance costs and downtime, ensuring devices remain up-to-date with the latest features and security patches. Additionally, bootloaders enhance security by implementing firmware encryption and authentication, ensuring that only authorized and verified firmware can be loaded onto the device.
In the event of firmware corruption or failure, a bootloader offers a critical recovery mechanism. It can detect issues with the main application firmware and enter a recovery mode to receive and

install a new, functional firmware image. This capability is essential for maintaining the reliability and availability of embedded systems, especially in mission-critical applications. Furthermore, bootloaders enable a modular and scalable firmware architecture by separating the bootloader and application code into distinct memory regions, facilitating easier and more flexible firmware management. This modular approach supports features like dual-boot configurations and rollback mechanisms, enhancing the overall robustness and scalability of the system.

Workflow

1. **Start:** Microcontroller is powered on or reset.
2. **Initialize Hardware:** Bootloader initializes system clock, memory, and peripherals.
3. **Check for Valid Firmware:** Bootloader checks if valid firmware is present in the application memory region.
4. **Valid Firmware Found:** If valid firmware is found, set the stack pointer and jump to the application.
5. **No Valid Firmware:** If no valid firmware is found, enter firmware update mode.
6. **Receive New Firmware:** Bootloader receives new firmware via a communication interface.
7. **Write Firmware to Memory:** Bootloader writes the new firmware to the application memory region.
8. **Verify Firmware:** Bootloader verifies the integrity of the new firmware.
9. **Jump to Application:** If the new firmware is valid, set the stack pointer and jump to the application.

Block Diagram

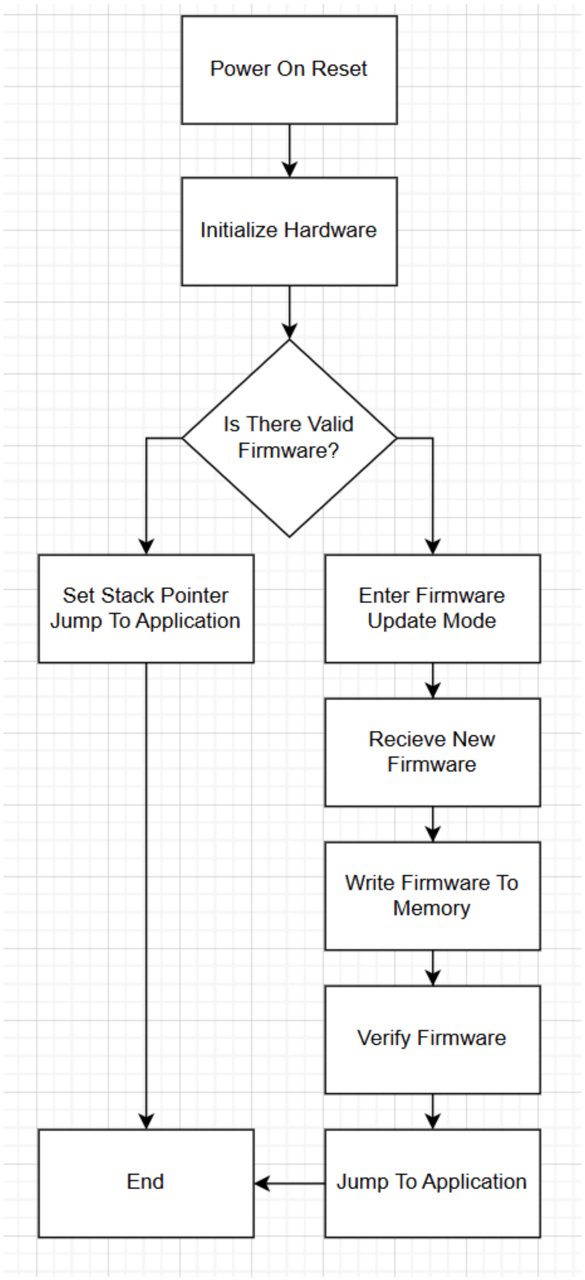

Figure 2: Block Diagram Of Workflow

Application Example

Bootloader (main.c)

- **Initialization**: Configures system clock, GPIO, and UART
- **Shared API**: Defines functions like Blink, TurnOn, and TurnOff in a shared section
- **Application Jump**: Checks for a valid application and jumps to it if found

```
#include "main.h"

#include

/* --- Type Definitions --- */

typedef void (*ptrF)(uint32_t dlyticks);

typedef void (*pFunction)(void);

struct BootloaderSharedAPI {

 void (*Blink)(uint32_t dlyticks);

 void (*TurnOn)(void);

 void (*TurnOff)(void);

};

/* --- Global Variables --- */

UART_HandleTypeDef huart2;

__attribute__((section(".myBufSectionRAM"))) unsigned char buf_ram[128];

__attribute__((section(".myBufSectionFLASH"))) const unsigned char buf_flash[10] =
{0,1,2,3,4,5,6,7,8,9};

#define LOCATE_FUNC __attribute__((section(".mysection")))

#define FLASH_APP_ADDR 0x8008000

/* --- Bootloader Shared API in Shared Section --- */

LOCATE_FUNC void Blink(uint32_t dlyticks) {
 HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
 HAL_Delay(dlyticks);
}
LOCATE_FUNC void TurnOn(void) {
 HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin, GPIO_PIN_SET);
}
LOCATE_FUNC void TurnOff(void) {
 HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin, GPIO_PIN_RESET);
}
__attribute__((section(".API_SHARED"))) struct BootloaderSharedAPI api = {
 .Blink = Blink,
 .TurnOn = TurnOn,
 .TurnOff = TurnOff
};
/* --- Function Prototypes --- */
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
void go2APP(void);
#ifdef FIRST_VIDEO
static ptrF Functions[] = { Blink };
#endif
/* --- Application Jump Handler --- */
void go2APP(void) {
 uint32_t JumpAddress;
 pFunction Jump_To_Application;
 printf("Bootloader Start\r\n");
 if (((*(uint32_t*)FLASH_APP_ADDR) & 0x2FFE0000) == 0x20000000) {
 printf("APP Start ...\r\n");
 HAL_Delay(100);
 JumpAddress = *(uint32_t*)(FLASH_APP_ADDR + 4);
 Jump_To_Application = (pFunction)JumpAddress;
 __set_MSP(*(uint32_t*)FLASH_APP_ADDR);
 Jump_To_Application();
 } else {
 printf("No APP found\r\n");
 }
}
int _write(int file, char *ptr, int len) {
 for (int i = 0; i < len; i++) {
 HAL_UART_Transmit(&huart2, (uint8_t *)ptr++, 1, 100);
 }
 return len;
}
__attribute__((section(".RamFunc"))) void TurnOnLED(GPIO_PinState PinState) {
 if (PinState != GPIO_PIN_RESET) {
 LED_GREEN_GPIO_Port->BSRR = LED_GREEN_Pin;
 } else {
 LED_GREEN_GPIO_Port->BRR = LED_GREEN_Pin;
 }
```

```
}

/* --- Main --- */

int main(void) {

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();

 MX_USART2_UART_Init();

 while (1) {

 #ifdef FIRST_VIDEO

 (*Functions[0])(100);

 #endif

 go2APP();

 }

}
```

Application (main.c)

- **Initialization**: Similar to the bootloader, initializes system clock, GPIO, and UART.
- **Main Loop**: Runs the application, utilizing the shared API functions.

```
/* Includes ------------------------------------------------------------------*/

#include "main.h"

#include "stdio.h"

/* Private typedef -----------------------------------------------------------*/

typedef void(*ptrF)(uint32_t dlyticks);

typedef void (*pFunction)(void);
```

```
/* Bootloader Shared API Struct */

struct BootloaderAPI {

 void (*Blink)(uint32_t dlyticks);

 void (*TurnOn)(void);

 void (*TurnOff)(void);

};

/* UART handle ---------------------------------------------------------------*/

UART_HandleTypeDef huart2;

/* Section variables ---------------------------------------------------------*/

unsigned char __attribute__((section(".myBufSectionRAM"))) buf_ram[128];

const unsigned char __attribute__((section(".myBufSectionFLASH"))) buf_flash[10] =
{0,1,2,3,4,5,6,7,8,9};

#define FLASH_APP_ADDR (0x8008000)

/* Function Prototypes -------------------------------------------------------*/

void SystemClock_Config(void);

static void MX_GPIO_Init(void);

static void MX_USART2_UART_Init(void);

/* Support printf over UART ---------------------------------------------
```

References

[1] “STMicroelectronics,” YouTube, https://www.youtube.com/@stmicroelectronics (accessed May 13, 2025).