Device Firmware Update (DFU) on STM32
A Practical Guide to Using DFU with STM32 Microcontrollers

Roy Ratcliffe[1,*]

Abstract

This article provides a practical guide to using Device Firmware Update (DFU) on STM32 microcontrollers. It covers the DFU protocol, how to set up the STM32 for DFU mode, and how to use DFU tools to update firmware. The article is intended for embedded systems developers who want to implement DFU in their projects.

Keywords: C, Embedded Systems, STM32

1. Device Firmware Update (DFU) on STM32

What is Device Firmware Update (DFU) mode? It is a special mode that allows you to update the firmware of a device without needing to use a bootloader. DFU is a protocol that will enable you to update the firmware of a device over USB or other communication interfaces. It is commonly used in embedded microcontrollers.

DFU mode is a built-in bootloader feature in many STM32 microcontrollers. The protocol is standardised and supported by tools like STM32CubeProgrammer. Many STM32 boards enter DFU mode by holding down the BOOT0 pin while resetting the device; however, not all STM32 families use this method. Some newer devices employ different mechanisms for entering DFU mode.

1.1. Why DFU?

DFU mode is helpful for several reasons:

1. **Ease of Use**: DFU mode simplifies the firmware update process by eliminating the need for a separate bootloader. This can make the development and deployment process more straightforward.
2. **Flexibility**: DFU mode enables firmware updates over various communication interfaces, including USB, UART, and others, allowing for greater flexibility in delivering firmware updates.
3. **Reliability**: The DFU protocol includes mechanisms for verifying the integrity of the firmware being uploaded, which can help ensure that updates are applied successfully and without corruption.
4. **Standardisation**: DFU is a standardised protocol, meaning various tools and libraries widely support it. This can make it easier to find resources and support for implementing DFU mode in your projects.

The DFU protocol is designed to be efficient and straightforward, enabling you to update the device's firmware quickly and easily. However, what if you want to implement DFU mode on your STM32 device without using a bootloader? This is where the `vJumpToDFU` function comes in. It allows you to enter DFU mode directly from your application code, without needing to use a bootloader.

[*]Corresponding author
Email address: `roy@ratcliffe.me` (Roy Ratcliffe)
[1]See more hackery at GitHub.

1.2. *How to Implement Manual DFU Mode on STM32*

Hopefully, the following code snippet will help you to implement DFU mode on your STM32 device quickly. It is a simple function that you can call to enter DFU mode. It disables interrupts, clears all interrupt enable and pending registers, sets the main stack pointer, and jumps to the DFU entry point. Here is the code snippet:

```c
/*!
 * \brief Initiates Device Firmware Update (DFU) mode.
 * \param pulMSP_PC Pointer to the DFU entry point address. The first element should
 * be the main stack pointer value, and the second element should be the
 * address of the DFU entry point.
 *
 * Disables interrupts, clears all interrupt enable and pending registers, sets
 * the main stack pointer, and jumps to the DFU entry point. Intended to be
 * called when a DFU command is received. The function does not return; it will
 * enter DFU mode and never return to the caller.
 *
 * \note Uses the SysTick control register to disable the SysTick timer, which
 * is typically used for system timing and task scheduling in FreeRTOS.
 * Disabling it is \b essential to prevent any further scheduling or timing
 * operations while in DFU mode!
 * \note Uses the NVIC (Nested Vectored Interrupt Controller) to clear all
 * interrupt enable and pending registers. This ensures that no interrupts can
 * occur when the core enters DFU mode.
 * \note Jumps to the DFU entry point, which is expected to be located at the
 * address provided in the \c pulMSP_PC parameter. The boot loader is responsible for
 * handling the actual firmware update process, including reading the new
 * firmware, writing it to the appropriate memory locations, and verifying its
 * integrity.
 * \note Enters an infinite loop after jumping to the DFU entry point in order
 * to prevent the execution of any further code if the DFU process returns.
 * \warning Should only be called when the system is ready to enter DFU mode.
 * It will disable all interrupts and clear all interrupt enable and pending
 * registers, which may lead to loss of data or state if called at an
 * inappropriate time. Ensure that all necessary data is saved and that the
 * system is in a safe state before calling this function. Your mileage may
 * vary.
 */
void vJumpToDFU(uint32_t *pulMSP_PC) __attribute__((noreturn));

void vJumpToDFU(uint32_t *pulMSP_PC) {
  /*
   * Disable interrupts upfront.
   * This is essential to prevent any further scheduling or interrupts pre-empting
   * while transitioning to DFU mode.
   * The inlined `__disable_irq` operation disables all interrupts, ensuring that no
   * further interrupts can occur while the core is entering DFU mode.
   */
  __disable_irq();

  /*
   * Disable SysTick timer.
   * This is essential to prevent any further scheduling or timing operations
   * while in DFU mode.
   * The SysTick timer is typically used for system timing and task scheduling
   * in FreeRTOS, and disabling it is essential to prevent any further
   * scheduling or timing operations while in DFU mode.
   */
  SysTick->CTRL = 0x00000000UL;

  /*
   * Clear all interrupt enable registers.
   * This ensures that no interrupts can occur when the core enters DFU mode.
   */
  WR_ALL_REGS(NVIC->ICER, 0xffffffffUL);

  /*
   * Clear all pending interrupts.
   * This ensures that no pending interrupts can occur when the core enters DFU
   * mode after re-enabling interrupts.
   */
  WR_ALL_REGS(NVIC->ICPR, 0xffffffffUL);

  /*
   * Enable interrupts again.
   * This is necessary to allow the boot loader to handle any interrupts that
   * may occur during the DFU process.
   * It does not automatically re-enable interrupts, as the
   * Cortex-M core's global interrupt enable bit resets to the unmasked state.
   * Note that this does not re-enable the SysTick timer, which remains disabled.
   */
  __enable_irq();

  /*
   * Set the main stack pointer to the value provided in the pul parameter.
   * This is necessary to ensure that the core starts executing code from the
   * correct stack.
   */
  __set_MSP(pulMSP_PC[0]);

  /*
   * Jump to the DFU entry point.
   * The boot loader is responsible for handling the actual firmware update
   * process, including reading the new firmware, writing it to the
   * appropriate memory locations, and verifying its integrity.
   */
  ((void (*)(void))(pulMSP_PC[1]))();

  /*
   * Enter an infinite loop after jumping to the DFU entry point.
   * This is necessary to prevent the execution of any further code if the
   * DFU process returns.
   * The boot loader is expected to handle the firmware update process and
   * will not return to this function.
```

```
   */
  while (1)
    ;
}
```

1.3. *Usage*

First, determine the correct indirect MSP and PC address for your STM32. On STM32F4, the indirect MSP and PC addresses are typically `0x20020000` and `0x08000000`, respectively. The first address is the main stack pointer (MSP) value, and the second address is the program counter (PC) value. The MSP value is the address of the top of the stack, and the PC value is the address of the DFU entry point. You can find these values in the STM32 reference manual.

Next, call the `vJumpToDFU` function with the correct MSP and PC values. For example, if the MSP value is `0x20020000` and the PC value is `0x08000000`, you can call the function like this:

```
#include <stdint.h> /* for uint32_t */

void vJumpToDFU(uint32_t *pulMSP_PC);

int main(void)
{
  uint32_t ulMSP_PC[2] = {0x20020000UL, 0x08000000UL};
  vJumpToDFU(ulMSP_PC);
  while (1);
}
```

This code snippet includes the `vJumpToDFU` function and a simple `main` function that calls it with the correct MSP and PC values. The `vJumpToDFU` function disables interrupts, clears all interrupt enable and pending registers, sets the main stack pointer, and jumps to the DFU entry point.

Alternatively, you can use the `vJumpToDFU` function to invoke the DFU indirectly if you know where your STM32 stores its DFU MSP-PC values. On the STM32F4, the DFU entry point is typically located at address `0x1FFF0000`. You can call the function like this:

```
vJumpToDFU((uint32_t *)0x1fff0000UL);
```

In the STM32's memory map, address $1FFF,0000_{16}$ is the start of system memory, where the bootloader resides.

1.4. *Writing to all registers in a block*

The `vJumpToDFU` function uses a macro called `WR_ALL_REGS` to write the same value to all registers in a block for convenience. This macro is defined below.

```
#include <stddef.h> /* for size_t */

/*!
 * \brief Write the same value to all registers in a block.
 * \details
 * This macro writes the same value to all registers in a contiguous block
 * of memory-mapped registers. It assumes that the registers are of the
 * same type and size.
 * \param _regs_ An array of registers.
 * \param _data_ The value to write to each register.
 */
#define WR_ALL_REGS(_regs_, _data_)                                 \
  do                                                                        \
    for (size_t addr = 0; addr < sizeof(_regs_) / sizeof((_regs_)[0]); addr++) \
      (_regs_)[addr] = (_data_);                                             \
  while (0)
```

The macro is used to assign the same value to all registers within a contiguous block of memory-mapped registers. This helps initialise or reset multiple registers to the same value.

The implementation makes certain assumptions about the type of registers; therefore, use it cautiously. Its design works with arrays of registers where each register has the same type and size. Consequently, it is not suitable for writing to individual registers or registers of different types. It is also unsuitable for registers that are not contiguous in memory, as it assumes that the registers form a contiguous block in memory. Additionally, it is not appropriate for registers of different types, since it assumes that all registers are of the same type and size.

The macro `WR_ALL_REGS` takes two arguments:

1. `_regs_`: an array of registers (e.g., `uint32_t regs[]`).
2. `_data_`: the value to write to each register.

The first argument is the array of registers, and the second argument is the value to write to each register.

It applies the ubiquitous `do { ... } while (0)` idiom to ensure that the macro behaves like a single statement, allowing it to be used in any context where a statement is expected. The construct requires a terminating semicolon when used, as it expands to a block of code that includes a loop.

References

[1] STMicroelectronics, Usb dfu protocol used in the stm32 bootloader, Tech. Rep. AN3156, STMicroelectronics, revision 14 (2023).
URL https://www.st.com/resource/en/application_note/cd00264379-usb-dfu-protocol-used-in-the-stm32-bootloader-stmicroelectronics.pdf