

Eine Anleitung zur Bare-Metal-Programmierung

Teil 1: STM32 und andere Controller

Von Sergey Lyubka (Irland)

Möchten Sie Mikrocontroller auf ihrer untersten Ebene programmieren, wo Sie ein tieferes Verständnis dafür bekommen können, wie sie tatsächlich funktionieren? Diese Anleitung für Entwickler hilft Ihnen bei den ersten Schritten, auf denen nur der GCC-Compiler und ein Referenzhandbuch verwendet werden. Die hier gelernten Grundlagen werden Ihnen helfen, besser zu verstehen, wie Frameworks wie Cube, Keil und Arduino ihre Arbeit verrichten. In dieser zweiteiligen Anleitung verwenden wir den STM32F429-Controller auf einem Nucleo-F429ZI-Board, aber was Sie hier lernen, kann leicht auch bei anderen Mikrocontrollern genutzt werden.

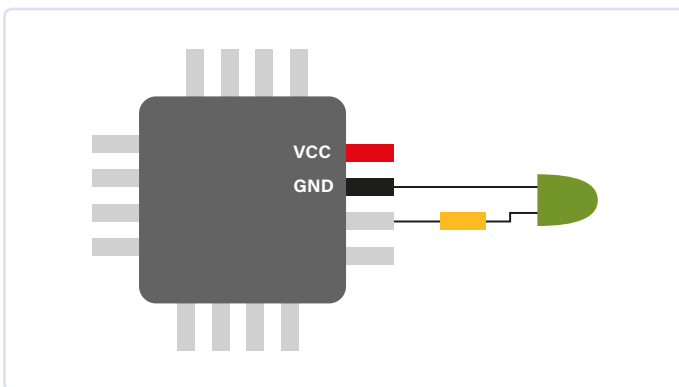


Bild 1. Der Firmware-Code kann eine hohe oder niedrige Spannung an einem Signal-Pin setzen, so dass eine LED blinkt.

Ein Mikrocontroller (μC oder MCU) ist ein kleiner Computer. Er verfügt in der Regel über eine CPU, einen Arbeitsspeicher (RAM), einen Flash-Speicher zum Speichern des Firmware-Codes und eine Reihe von Anschlüssen, auf die man von außen zugreifen kann. Einige dieser Anschlüsse sind für die Stromversorgung der MCU vorgesehen und in der Regel als GND (Masse) und VCC gekennzeichnet. Andere Pins werden zur Kommunikation mit der MCU verwendet, wobei eine hohe oder niedrige Spannung an diese Anschlüsse gelegt wird. Eines der einfachsten Kommunikationsmittel ist eine an einen Pin angeschlossene LED: Der eine LED-Kontakt ist mit dem Masse-Pin (GND) verbunden, der andere Kontakt über einen Strombegrenzungswiderstand mit einem Signal-Pin der MCU. Der Firmware-Code kann eine hohe oder niedrige Spannung an diesen Signal-Pin einstellen, wodurch die LED blinkt (**Bild 1**).

Erforderliche Hardware und Softwaretools

In dieser Anleitung verwenden wir ein Nucleo-F429ZI Entwicklungsboard (erhältlich bei Mouser und anderen Händlern). Um diesem Tutorial zu folgen, laden Sie das Referenzhandbuch der STM32F429-MCU [1] und dann das Benutzerhandbuch für das Entwicklungsboard [2] herunter.

Weiterhin werden die folgenden Softwaretools benötigt:
ARM GCC, <https://launchpad.net/gcc-arm-embedded> - zum Kompilieren und Linken
GNU make, <https://gnu.org/software/make> - für die Build-Automatisierung
ST link, <https://github.com/stlink-org/stlink> - zum Flashen

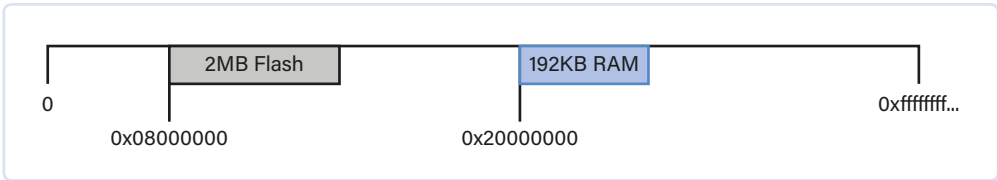
Wir zeigen hier die Installationsanweisungen für Linux (Ubuntu).

Starten Sie ein Terminal, dann führen Sie aus:

```
$ sudo apt -y update
$ sudo apt -y install gcc-arm-none-eabi make
  stlink-tools
```

Um die Tools auf einem Mac oder Windows-PC einzurichten, siehe [3].

Bild 2. Flash- und RAM-Speicherbereiche des STM32F429.



Speicher und Register

Der Adressraum einer 32-Bit-MCU wie dem STM32F429 von STMicroelectronics ist in Bereiche unterteilt. Ein Speicherbereich (an einer bestimmten Adresse) ist beispielsweise auf den internen MCU-Flashspeicher abgebildet. Firmware-Anweisungen werden aus diesem Speicherbereich gelesen und ausgeführt. Ein weiterer Bereich ist das RAM, das ebenfalls einer bestimmten Adresse zugeordnet ist. Wir können beliebige Werte im RAM-Bereich lesen und dorthin schreiben.

Im Referenzhandbuch des STM32F429-[1] können wir in Abschnitt 2.3.1 nachsehen, dass der RAM-Bereich bei Adresse 0x20000000 beginnt und 192 KB groß ist. Aus Abschnitt 2.4 erfahren wir, dass der Flash-Speicher an der Adresse 0x08000000 abgebildet wird. Unsere MCU hat 2 MB Flash, also sind die Flash- und RAM-Bereiche wie in **Bild 2** angeordnet.

Aus dem Handbuch erfahren wir auch, dass es noch viele weitere Speicherbereiche gibt. Ihre Adressbereiche sind im Abschnitt 2.3, „Memory map“ angegeben. So gibt es zum Beispiel einen „GPIOA“-Bereich, der bei 0x40020000 beginnt und 1 KB groß ist. Diese Speicherbereiche entsprechen verschiedenen „Peripheriegeräten“ innerhalb der MCU. Das sind Teile der Siliziumschaltung, die bestimmte Pins auf eine besondere Weise reagieren lässt. Ein peripherer Speicherbereich ist eine Ansammlung von 32-Bit-Registern.

Jedes Register ist ein 4-Byte-Speicherbereich an einer bestimmten Adresse, die einer bestimmten Funktion der jeweiligen Peripherie zugeordnet ist. Durch das Schreiben von Werten in ein Register - mit anderen Worten, durch das Schreiben eines 32-Bit-Wertes an eine bestimmte Speicheradresse - können wir steuern, wie sich ein bestimmtes Peripheriegerät verhalten soll. Durch das Lesen dieser Register können wir die Daten oder die Konfiguration eines Peripheriegeräts zurückerhalten.

Es gibt viele verschiedene Peripheriegeräte. Eines der einfacheren ist GPIO (General Purpose Input Output), das es dem User ermöglicht, MCU-Pins in einen „Ausgabemodus“ zu versetzen und eine hohe oder niedrige Spannung an ihnen einzustellen. Oder er kann für Pins in einen „Eingangsmodus“ einstellen und Spannungswerte von ihnen lesen. Es gibt einen UART-Peripheriebaustein, der Daten gemäß eines seriellen Protokolls über zwei Pins senden und empfangen kann.

Oft gibt es mehrere „Instanzen“ desselben Peripheriegeräts, zum Beispiel GPIOA, GPIOB und so weiter, die verschiedene Gruppen von MCU-Pins steuern. Ebenso kann es UART1, UART2 und so weiter geben, die die Implementierung mehrerer UART-Kanäle ermöglichen. Auf dem STM32F429 gibt es mehrere GPIO- und UART-Peripherieinstanzen.

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

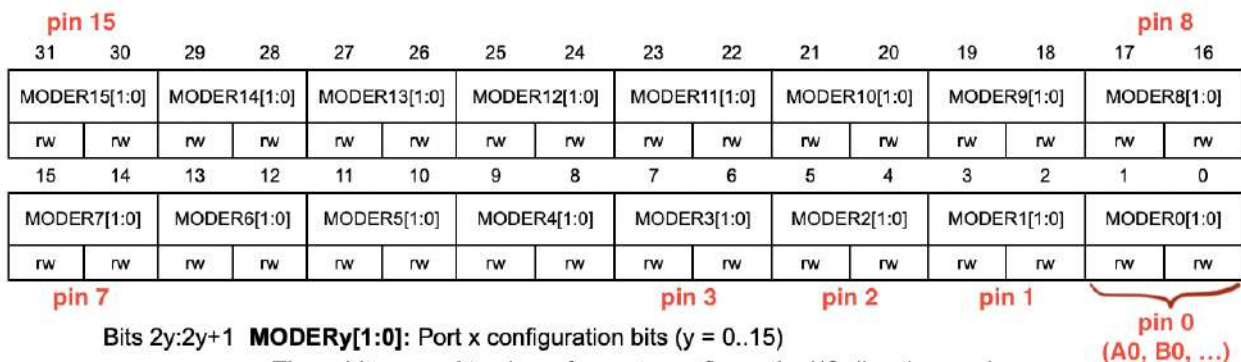


Bild 3. Die Beschreibung der GPIO-Register finden Sie im Referenzhandbuch. Ein MODER-Register steuert 16 physikalische Pins. (Quelle: [1])

Die GPIOA-Peripherie zum Beispiel beginnt bei 0x40020000, und die Beschreibung der GPIO-Register finden Sie in Abschnitt 8.4 [1]. Das Referenzhandbuch sagt, dass das `GPIOA_MODER`-Register den Offset 0 hat, was bedeutet, dass seine Adresse `0x40020000 + 0` ist. In **Bild 3** können wir das Format des Registers sehen.

Das Handbuch zeigt, dass das 32-Bit-`MODER`-Register eine 16 Bit breite Sammlung von 2-Bit-Werten ist. Ein `MODER`-Register steuert 16 physische Pins: Die Bits 0...1 steuern Pin 0, Bits 2...3 steuern Pin 1 und so weiter. Der 2-Bit-Wert kodiert den Pin-Modus: 0 bedeutet Eingang, 1 bedeutet Ausgang, 2 bedeutet „alternative Funktion“ - ein bestimmtes Verhalten, das an anderer Stelle beschrieben wird, und 3 bedeutet analog. Da der Name des Peripheriegeräts `GPIOA` lautet, werden die Pins als „A0“, „A1“ ... „Ax“ bezeichnet. Für die `GPIOB`-Peripherie würde die Pin-Bezeichnung „B0“, „B1“ ... lauten. Wenn wir den 32-Bit-Wert 0 in das `MODER`-Register schreiben, versetzen wir alle 16 Pins von A0 bis A15 in den Eingangsmodus:

```
* (volatile uint32_t *) (0x40020000 + 0) = 0;
// Set A0...A15 to input mode
```

Beachten Sie Qualifizierer `volatile`, dessen Bedeutung später behandelt wird. Durch das Setzen einzelner Bits können wir bestimmte Pins selektiv auf einen gewünschten Modus einstellen. Folgendes Snippet setzt zum Beispiel Pin A3 in den Ausgabemodus:

```
* (volatile uint32_t *) (0x40020000 + 0) &= ~(3 << 6);
// Clear bit range 6...7
* (volatile uint32_t *) (0x40020000 + 0) |= 1 << 6;
// Set bit range 6...7 to 1
```

Lassen Sie mich diese Bitoperationen erklären. Ziel ist es, die Bits 6...7, die für Pin 3 der GPIOA-Peripherie zuständig sind, auf einen bestimmten Wert (in unserem Fall 1) zu setzen. Dies geschieht in zwei Schritten. Zunächst muss der aktuelle Wert der Bits 6...7 gelöscht werden, da sie ja bereits einen Wert enthalten könnten. Dann müssen wir die entsprechenden Bits von 6...7 setzen, um den gewünschten Wert zu erhalten.

Zuerst müssen wir also den Bitbereich 6...7 (zwei Bits an Position 6) auf 0 setzen. Wie man eine Anzahl von Bits auf 0 setzt, zeigen die vier Schritte in **Tabelle 1**.

Tabelle 1. Setzen bestimmter Bits auf 0

Aktion	Ausdruck	Bits (erste 12 von 32)
Holt eine Zahl, bei der N zusammenhängende Bits gesetzt sind: 2^N-1 , hier mit $N=2$	3	000000000011
Verschiebe die Zahl um X Stellen nach links	$(3 << 6)$	000011000000
Invertiere die Zahl: Verwandle Nullen in Einsen und Einsen in Nullen	$\sim(3 << 6)$	111100111111
Logisches UND mit bestehendem Wert	<code>VAL &= $\sim(3 << 6)$</code>	xxxxx00xxxxxx

Beachten Sie, dass die letzte Operation, das logische UND, N Bits an der Position X auf 0 setzt (weil sie mit 0 UND-verknüpft werden), aber den Wert aller anderen Bits beibehält (weil sie mit 1 UND-verknüpft werden). Die Beibehaltung des bestehenden Wertes ist wichtig, weil wir die Einstellungen in anderen Bitbereichen nicht ändern wollen. Wenn wir also im Allgemeinen N Bits an der Position X löschen wollen, können wir folgendes schreiben:

```
REGISTER &= ~(2^N - 1) << X);
```

Und schließlich wollen wir einen bestimmten Bitbereich auf den gewünschten Wert setzen. Wir verschieben diesen Wert um X Positionen nach links und verknüpfen ihn mit dem aktuellen Wert des gesamten Registers (um die Werte der anderen Bits zu erhalten):

```
REGISTER |= VALUE << X;
```

Von Menschen lesbare Programmierung von Peripheriegeräten

Im vorigen Abschnitt haben wir gelernt, dass wir ein Peripherieregister lesen und schreiben können, indem wir direkt auf bestimmte Speicheradressen zugreifen. Schauen wir uns das Snippet an, das Pin A3 in den Ausgabemodus versetzt:

```
* (volatile uint32_t *) (0x40020000 + 0) &= ~(3 << 6);
// Clear bit range 6...7
* (volatile uint32_t *) (0x40020000 + 0) |= 1 << 6;
// Set bit range 6...7 to 1
```

Das ist ziemlich kryptisch. Ohne ausführliche Kommentare wäre ein solcher Code ziemlich schwer zu verstehen. Wir können diesen Code in eine viel besser lesbare Form umschreiben. Die Idee dabei ist, das gesamte Peripheriegerät als eine Struktur darzustellen, die 32-Bit-Felder enthält. Schauen wir uns an, welche Register für die GPIO-Peripherie in Abschnitt 8.4 des Referenzhandbuchs vorhanden sind. Es sind `MODER`, `OTYPER`, `OSPEEDR`, `PUPDR`, `IDR`, `ODR`, `BSRR`, `LCKR`, `AFR`. Ihre Offsets sind 0, 4, 8, und so weiter. Das bedeutet, dass wir sie als eine Struktur mit 32-Bit-Feldern darstellen und ein `#define` für GPIOA erstellen können:

```
struct gpio {
    volatile uint32_t MODER, OTYPER, OSPEEDR,
        PUPDR, IDR, ODR, BSRR, LCKR, AFR[2];
};
#define GPIOA ((struct gpio *) 0x40020000)
```

Dann können wir für die Einstellung des GPIO-Pin-Modus eine Funktion definieren:

```
// Enum values are per reference manual: 0, 1, 2, 3
enum {GPIO_MODE_INPUT, GPIO_MODE_OUTPUT,
GPIO_MODE_AF, GPIO_MODE_ANALOG};

static inline void gpio_set_mode
(struct gpio *gpio, uint8_t pin, uint8_t mode) {
    gpio->MODER &= ~(3U << (pin * 2));
```

```

// Clear existing setting
gpio->MODER |= (mode & 3) << (pin * 2);
// Set new mode
}

```

Jetzt können wir das Snippet für A3 wie folgt umschreiben:

```

gpio_set_mode(GPIOA, 3 /* pin */, GPIO_MODE_OUTPUT);
// Set A3 to output

```

Unsere MCU hat mehrere GPIO-Peripheriegeräte (auch „Bänke“ genannt): A, B, C, ... K. Aus Abschnitt 2.3 wissen wir, dass sie 1 KB voneinander entfernt sind: GPIOA befindet sich an der Adresse 0x40020000, GPIOB an 0x40020400, und so weiter:

```

#define GPIO(bank) ((struct gpio *)
(0x40020000 + 0x400 * (bank)))

```

Wir können eine Pin-Nummerierung definieren, die die Bank und die Pin-Nummer enthält. Dazu verwenden wir einen 2-Byte-Wert `uint16_t`, wobei das obere Byte die GPIO-Bank und das untere Byte die Pin-Nummer angibt:

```

#define PIN(bank, num) (((bank) - 'A') << 8) | (num))
#define PINNO(pin) (pin & 255)
#define PINBANK(pin) (pin >> 8)

```

Auf diese Weise können wir Pins für jede GPIO-Bank festlegen:

```

uint16_t pin1 = PIN('A', 3); // A3 - GPIOA pin 3
uint16_t pin2 = PIN('G', 11); // G11 - GPIOG pin 11

```

Schreiben wir die Funktion `gpio_set_mode()` neu, um unsere Pin-Spezifikation zu übernehmen:

```

static inline void gpio_set_mode(uint16_t pin, uint8_t
mode) {
    struct gpio *gpio = GPIO(PINBANK(pin));
    // GPIO bank
    uint8_t n = PINNO(pin); // Pin number
    gpio->MODER &= ~(3U << (n * 2));
    // Clear existing setting
    gpio->MODER |= (mode & 3) << (n * 2);
    // Set new mode
}

```

Der Code für A3 ist nun selbsterklärend:

```

uint16_t pin = PIN('A', 3); // Pin A3
gpio_set_mode(pin, GPIO_MODE_OUTPUT); // Set to output

```

Wir haben hiermit eine nützliche erste API für die GPIO-Peripherie geschaffen. Andere Peripheriegeräte wie der UART für serielle Kommunikation und andere können auf ähnliche Weise implementiert werden. Dies ist eine gute Programmierpraxis, da es den Code selbsterklärend und für den Menschen lesbar macht.

MCU-Boot und Vektortabelle

Wenn eine ARM-MCU bootet, liest sie eine sogenannte „Vektortabelle“ vom Anfang des Flash-Speichers. Eine Vektortabelle ist ein gemeinsames Konzept für alle ARM-MCUs, ein Array von 32-Bit-Adressen von Interrupt-Handlern. Die ersten 16 Einträge sind von ARM reserviert und gelten für alle ARM-MCUs. Die restlichen Interrupt-Handler sind spezifisch für die jeweilige MCU, und dabei handelt es sich um Interrupt-Handler für Peripheriegeräte. Einfachere MCUs mit wenigen Peripheriegeräten haben wenige Interrupt-Handler, komplexere MCUs haben viele.

Die Vektortabelle für den STM32F429 ist in Tabelle 62 des Referenzhandbuchs [1] dokumentiert. Dort erfahren wir, dass es zusätzlich zu den 16 Standard-Handlern 91 Peripherie-Handler gibt.

Jeder Eintrag in der Vektortabelle ist eine Adresse einer Funktion, die die MCU ausführt, wenn ein Hardware-Interrupt (IRQ) ausgelöst wird. Eine Ausnahme sind die ersten beiden Einträge, die eine Schlüsselrolle im Boot-Prozess der MCU spielen. Bei diesen beiden ersten Werten handelt es sich um einen initialen Stapel-Pointer und die Adresse der Boot-Funktion (ein auszuführender Einstiegsunkt der Firmware).

Wir wissen nun, dass unsere Firmware so aufgebaut sein muss, dass der zweite 32-Bit-Wert im Flash die Adresse unserer Boot-Funktion enthält. Wenn die MCU bootet, liest sie diese Adresse aus dem Flash und springt zu unserer Boot-Funktion.

Minimale Firmware

Erstellen wir eine Datei `main.c` und spezifizieren unsere Boot-Funktion, die zunächst nichts tut (in eine Endlosschleife fällt), und zusätzlich eine Vektortabelle, die 16 Standardeinträge und 91 STM32-Einträge enthält. Erstellen und öffnen Sie in einem Editor Ihrer Wahl die Datei `main.c` und geben Sie Folgendes ein:

```

// Startup code
__attribute__((naked, noreturn)) void _reset(void) {
    for (;;) (void) 0; // Infinite loop
}

extern void _estack(void); // Defined in link.ld

// 16 standard and 91 STM32-specific handlers
__attribute__((section(".vectors")))
void (*tab[16 + 91])(void) = {_estack, _reset};

```

Für die Funktion `_reset()` haben wir die GCC-spezifischen Attribute `naked` und `noreturn` verwendet, was bedeutet, dass die Standardfunktionen `prologue` und `epilogue` vom Compiler nicht erstellt werden sollen und dass die Funktion nicht zurückkehrt.

Der Ausdruck `void (*tab[16 + 91])(void)` bedeutet: Definiere ein Array von 16 + 91 Pointern auf Funktionen, die nichts zurückgeben (`void`) und zwei Argumente annehmen. Jede solche Funktion ist ein IRQ-Handler (Interrupt ReQuest Handler). Ein Array dieser Handler wird als Vektortabelle bezeichnet.

Die Vektortabelle `tab` legen wir in einem separaten Abschnitt namens `.vectors` ab, weil wir ihn später brauchen, um dem Linker mitzuteilen, dass er diesen Abschnitt direkt an den Anfang der generierten Firmware setzen soll, und zwar fortlaufend, am Anfang

des Flash-Speichers. Die ersten beiden Einträge sind der Wert des Stack-Pointer-Registers und der Einstiegspunkt der Firmware. Den Rest der Vektortabelle lassen wir mit Nullen gefüllt.

Kompilierung

Lassen Sie uns unseren Code kompilieren. Starten Sie ein Terminal (oder einen Eingabe-Prompt in Windows) und führen Sie aus:

```
$ arm-none-eabi-gcc -mcpu=cortex-m4 main.c -c
```

Das funktioniert! Die Kompilierung hat eine Datei *main.o* erzeugt, die unsere minimale Firmware enthält, die nichts tut. Die Datei *main.o* liegt im ELF-Binärformat vor und enthält mehrere Abschnitte. Sehen wir uns diese in **Listing 1** an.

Die VMA/LMA-Adressen für die Abschnitte sind auf 0 gesetzt, was bedeutet, dass *main.o* noch keine vollständige Firmware ist, da sie keine Informationen darüber enthält, wo diese Abschnitte in den Adressraum geladen werden sollen. Wir müssen einen Linker verwenden, um aus *main.o* die vollständige Firmware-Datei *firmware.elf* zu erzeugen.

Der Abschnitt *.text* enthält Firmware-Code, in unserem Fall nur eine *_reset()*-Funktion, die zwei Bytes lang ist - eine Sprunganweisung zu ihrer eigenen Adresse. Es gibt weder einen leeren *.data*-Abschnitt noch einen leeren *.bss*-Abschnitt [8] (für nicht initialisierte, aber deklarierte Variablen wird dieser Abschnitt normalerweise mit Nullen gefüllt). Unsere Firmware wird in den Flash-Bereich an Offset 0x8000000 kopiert, aber unser Datenbereich sollte sich im RAM befinden - daher sollte unsere *_reset()*-Funktion den Inhalt des *.data*-Bereichs ins RAM kopieren. Außerdem muss sie Nullen in den gesamten *.bss*-Abschnitt schreiben. In unserem Fall sind die Abschnitte *.data* und *.bss* leer, aber wir müssen unsere *_reset()*-Funktion trotzdem modifizieren, um sie richtig anzuwenden.

Dazu müssen wir wissen, wo der Stack beginnt, und wo die Abschnitte *data* und *bss* beginnen. Dies können wir im *Linker-Skript* angeben, einer Datei mit Anweisungen an den Linker, wo verschiedene Abschnitte im Adressraum zu platzieren und welche Symbole zu erstellen sind.

Linker-Skript

Erstellen Sie eine Datei namens *link.ld* und kopieren Sie den Inhalt aus [4] hinein. Unten finden Sie die Erklärung:

```
ENTRY(_reset);
```

Diese Zeile teilt dem Linker den Wert des „entry point“-Attributs im generierten ELF-Header mit - dies ist also ein Duplikat dessen, was eine Vektortabelle enthält. Dies ist ein Hilfsmittel für einen Debugger (wie Ozone, beschrieben im zweiten Teil dieser Artikelreihe), das uns hilft, einen Haltepunkt am Anfang der Firmware zu setzen. Ein Debugger kennt keine Vektortabelle, daher ist er auf den ELF-Header angewiesen.

```
MEMORY {
flash(rx) : ORIGIN = 0x08000000, LENGTH = 2048k
sram(rwx) : ORIGIN = 0x20000000, LENGTH = 192k
/* remaining 64k in a separate address space */
}
```

Diese Zeilen teilen dem Linker mit, dass wir zwei Speicherbereiche sowie deren Adressen und Größen im Adressraum haben.

```
_estack = ORIGIN(sram) + LENGTH(sram);
/* stack points to end of SRAM */
```

Dies weist den Linker an, ein Symbol *_estack* zu erstellen, das einen Wert am Ende des RAM-Speicherbereichs enthält. Das wird unser anfänglicher Stack-Wert sein!

```
.vectors : { KEEP(*(.vectors)) } > flash
.text : { *(.text*) } > flash
.rodata : { *(.rodata*) } > flash
```

Diese Zeilen weisen den Linker an, zuerst die Vektortabelle im Flash abzulegen, gefolgt von der *.text*-Sektion (Firmware-Code), gefolgt von den Nur-Lese-Daten *.rodata*. Als nächstes kommt die *.data*-Sektion:

Listing 1: Kompilieren von main.o

```
$ arm-none-eabi-objdump -h main.o
```

```
...
Idx Name           Size      VMA      LMA      File off  Algn
  0 .text           00000002 00000000 00000000 00000034 2**1
CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data           00000000 00000000 00000000 00000036 2**0
CONTENTS, ALLOC, LOAD, DATA
  2 .bss            00000000 00000000 00000000 00000036 2**0
ALLOC
  3 .vectors        000001ac 00000000 00000000 00000038 2**2
CONTENTS, ALLOC, LOAD, RELOC, DATA
...
```

Listing 2: Startup-Code

```
int main(void) {
    return 0; // Do nothing so far
}

// Startup code
__attribute__((naked, noreturn)) void _reset(void) {
    // memset .bss to zero, and copy .data section to RAM region
    extern long _sbss, _ebss, _sdata, _edata, _sidata;
    for (long *src = &_sbss; src < &_ebss; src++) *src = 0;
    for (long *src = &_sdata, *dst = &_sidata; src < &_edata;) *src++ = *dst++;

    main(); // Call main()
    for (;;) (void) 0; // Infinite loop in the case if main() returns
}
```

```
.data : {
    _sdata = .; /* .data section start */
    *(.first_data)
    *(.data SORT(.data.*))
    _edata = .; /* .data section end */
} > sram AT > flash
_sidata = LOADADDR(.data);
```

Wir weisen den Linker an, die Symbole `_sdata` und `_edata` zu erstellen, die wir verwenden, um den Datenbereich in der `_reset()`-Funktion in das RAM zu kopieren. Dasselbe gilt für den `.bss`-Abschnitt:

```
.bss : {
    _sbss = .; /* .bss section start */
    *(.bss SORT(.bss.*) COMMON)
    _ebss = .; /* .bss section end */
} > sram
```

Startup-Code

Jetzt können wir unsere Funktion `_reset()` aktualisieren. Wir kopieren den `.data`-Abschnitt in das RAM und initialisieren den `.bss`-Abschnitt mit Nullen. Dann rufen wir die Funktion `main()` auf - und fallen in eine Endlosschleife, wenn `main()` zurückkehrt; siehe **Listing 2**. Das Diagramm in **Bild 4** zeigt, wie `_reset()` sowohl `.data` als auch `.bss` initialisiert.

Die Datei `firmware.bin` ist lediglich eine Verkettung der drei Abschnitte `.vectors` (IRQ-Vektortabelle), `.text` (Code) und `.data` (Daten). Diese Abschnitte wurden gemäß dem Linker-Skript aufgebaut: `.vectors` liegt ganz am Anfang des Flash-Speichers, `.text` folgt unmittelbar danach, und `.data` liegt weit oben. Die Adressen in `.text` liegen im Flash-Speicherbereich, die Adressen in `.data` im RAM-Bereich. Wenn eine Funktion eine Adresse hat, zum Beispiel `0x8000100`, dann befindet sie sich genau an dieser Adresse im Flash. Wenn der Code jedoch auf eine Variable im `.data`-Bereich über die Adresse zugreift, beispielsweise `0x20000200`, dann befindet sich an dieser Adresse nichts, denn beim Booten befindet sich der `.data`-Bereich in der Datei `firmware.bin` im Flash! Deshalb muss der Startup-Code den `.data`-Abschnitt aus dem Flash-Speicherbereich in den RAM-Bereich verschieben. Jetzt sind wir so weit, um die vollständige Firmware-Datei `firmware.elf` zu erzeugen:

```
$ arm-none-eabi-gcc -T link.ld -nostdlib main.o -o firmware.elf
```

Schauen wir uns in **Listing 3** die Abschnitte in `firmware.elf` an. Wir können sehen, dass der Abschnitt `.vectors` ganz am Anfang des Flash-Speichers an der Adresse `0x8000000` liegt, und der Abschnitt `.text` direkt danach an `0x80001ac`. Unser Code erzeugt keine Variablen, daher gibt es auch keinen Datenbereich.

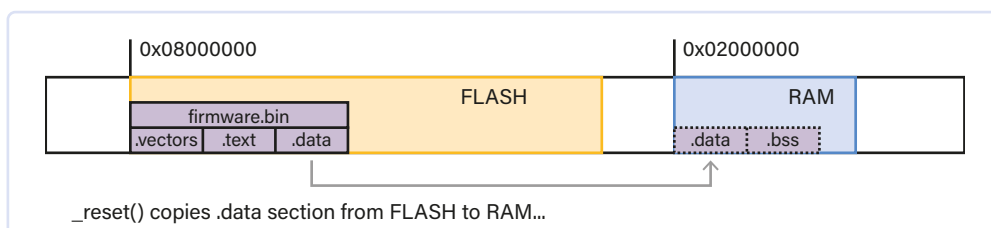


Bild 4. Das Diagramm veranschaulicht, wie `_reset()` `.data` und `.bss` initialisiert.



Listing 3: Bereiche in firmware.elf

000011000000

```
$ arm-none-eabi-objdump -h firmware.elf
...
Idx Name           Size      VMA      LMA      File off  Algn
  0 .vectors         000001ac  08000000  08000000  00010000  2**2
                CONTENTS, ALLOC, LOAD, DATA
  1 .text            00000058  080001ac  080001ac  000101ac  2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
...
```

Firmware flashen

Nun können wir die Firmware zu flashen! Zuerst extrahieren wir die Abschnitte aus *firmware.elf* in einen einzelnen, zusammenhängenden Binärblob:

```
$ arm-none-eabi-objcopy -O binary firmware.elf firmware.bin
```

Schließen Sie Ihr Board an den USB an und verwenden Sie das Dienstprogramm *st-link*, um *firmware.bin* zu flashen:

```
$ st-flash --reset write firmware.bin 0x8000000
```

Geschafft! Wir haben eine Firmware geflasht, die nichts tut.

Build-Automatisierung mit Makefile

Anstatt all diese Befehle zum Kompilieren, Linken und Flashen einzugeben, können wir das Kommandozeilenwerkzeug *make* verwenden, um den gesamten Prozess zu automatisieren. Das Dienstprogramm *make* verwendet eine Konfigurationsdatei namens Makefile, aus der es Anweisungen für die Ausführung von Aktionen liest. Diese Automatisierung ist großartig, weil sie auch den Prozess der Firmware-Erstellung, die verwendeten Kompilierungsflags und so weiter dokumentiert.

Es gibt für alle, die neu in *make* sind, ein großartiges Makefile-Tutorial [5]. Im Folgenden führe ich die wichtigsten Konzepte auf, die zum Verständnis unseres einfachen Bare-Metal-Makefiles erforderlich sind. Wer mit *make* bereits vertraut ist, kann ruhigen Gewissens diesen Abschnitt überspringen. Das Makefile-Format ist einfach:

```
action1:
    command ... # Comments can go after hash symbol
    command ... # IMPORTANT: command must be preceded
                with the TAB character
action2:
    command ... # Don't forget about TAB. Spaces won't
                work!
```

Jetzt können wir *make* mit dem Namen der Aktion (auch *target* genannt) aufrufen, um eine entsprechende Aktion auszuführen:

```
$ make action1
```

Es ist möglich, Variablen zu definieren und sie in Befehlen zu verwenden. Außerdem können Aktionen die Namen von Dateien sein, die erstellt werden müssen:

```
firmware.elf:
    COMPILATION COMMAND .....
```

Außerdem kann jede Aktion eine Liste von Abhängigkeiten haben. Zum Beispiel hängt *firmware.elf* von unserer Quelldatei *main.c* ab. Immer wenn sich die Datei *main.c* ändert, wird *firmware.elf* mit dem Befehl *make build* neu erstellt:

```
build: firmware.elf
firmware.elf: main.c
    COMPILATION COMMAND
```

Schreiben wir nun also ein Makefile für unsere Firmware. Wir definieren eine *build*-Aktion/Target:

```
CFLAGS ?= -W -Wall -Wextra -Werror -Wundef -Wshadow
-Wdouble-promotion \ -Wformat-truncation -fno-common
-Wconversion \ -g3 -Os -ffunction-sections -fdata-sections
-I. \ -mcpu=cortex-m4 -mthumb -mfloat-abi=hard
-mfpu=fpv4-sp-d16 $(EXTRA_CFLAGS)
LDFLAGS ?= -Tlink.ld -nostartfiles -nostdlib --specs nano.
specs -lc -lgcc -Wl,--gc-sections -Wl,-Map=$@.map
SOURCES = main.c
build: firmware.elf
firmware.elf: $(SOURCES)
    arm-none-eabi-gcc $(SOURCES) $(CFLAGS) $(LDFLAGS)
    -o $@
```

Hier definieren wir Flags zum Kompilieren. Das *?* steht für einen Standardwert, der auf der Kommandozeile wie folgt überschrieben werden kann:

```
$ make build CFLAGS="-O2 ...."
```

Wir geben die Variablen *CFLAGS*, *LDFLAGS* und *SOURCES* an. Dann teilen wir *make* mit: Wenn dir gesagt wird, dass du „builden“ sollst, dann erstelle eine Datei *firmware.elf*. Sie hängt von der Datei *main.c* ab, und um sie zu erstellen, starte den Compiler *arm-none-eabi-gcc* mit den angegebenen Flags. Die spezielle Variable *\$@* expandiert zu einem Zielnamen - in unserem Fall *firmware.elf*. Rufen wir nun *make* auf:

```
$ make build arm-none-eabi-gcc main.c -W -Wall -Wextra
-Werror -Wundef -Wshadow -Wdouble-promotion -Wformat-truncation
-fno-common -Wconversion -g3 -Os -ffunction-sections -fdata-sections
-I. -mcpu=cortex-m4 -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16 -Tlink.ld
-nostartfiles -nostdlib --specs nano.specs -lc -lgcc -Wl,--gc-sections
-Wl,-Map=firmware.elf.map -o firmware.elf
```

Wir führen es erneut aus:



Listing 4: Ausschnitt der main.c-Datei Blinky LED

```
#include <inttypes.h>
#include <stdbool.h>
#define BIT(x) (1UL << (x))
#define PIN(bank, num) (((bank) - 'A') << 8) | (num)
#define PINNO(pin) (pin & 255)
#define PINBANK(pin) (pin >> 8)

struct gpio {
    volatile uint32_t MODER, OYPER, OSPEEDR, PUPDR, IDR, ODR, BSRR, LCKR, AFR[2];
};
#define GPIO(bank) ((struct gpio *) (0x40020000 + 0x400 * (bank)))

// Enum values are per datasheet: 0, 1, 2, 3
enum { GPIO_MODE_INPUT, GPIO_MODE_OUTPUT, GPIO_MODE_AF, GPIO_MODE_ANALOG };

static inline void gpio_set_mode(uint16_t pin, uint8_t mode) {
    struct gpio *gpio = GPIO(PINBANK(pin)); // GPIO bank
    int n = PINNO(pin); // Pin number
    gpio->MODER &= ~(3U << (n * 2)); // Clear existing setting
    gpio->MODER |= (mode & 3) << (n * 2); // Set new mode
}
```

```
$ make build
make: Nothing to be done for 'build'.
```

Das Dienstprogramm `make` prüft die Änderungen bei der Abhängigkeit `main.c` und `firmware.elf` - und unternimmt nichts, wenn `firmware.elf` auf dem neuesten Stand ist. Aber wenn wir `main.c` ändern, kompiliert das nächste `make build` neu:

```
$ touch main.c # Simulate changes in main.c
$ make build
```

Jetzt ist nur noch das `flash`-Target übrig:

```
firmware.bin: firmware.elf
$(DOCKER) $(CROSS)-objcopy -O binary $< $@ flash:
firmware.bin
st-flash --reset write $(TARGET).bin 0x8000000
```

Das war's! Jetzt erstellt der Terminalbefehl `make flash` eine Datei `firmware.bin` und flasht sie auf das Board. Die Firmware wird neu kompiliert, wenn `main.c` geändert wird, weil `firmware.bin` von `firmware.elf` abhängt, und diese wiederum von `main.c`. Der Entwicklungszyklus würde also aus diesen beiden Aktionen in einer Schleife bestehen:

```
# Develop code in main.c
$ make flash
```

Es ist eine gute Idee, einen Clean-Befehl hinzuzufügen, um Build-Artefakte zu entfernen:

```
clean:
    rm -rf firmware.*
```

Den vollständigen Quellcode des Projekts finden Sie im Ordner *Step 0 minimal* [6].

Blinkende LED

Nun, da wir die gesamte Build-/Flash-Infrastruktur eingerichtet haben, können wir unserer Firmware auch etwas Nützliches beizubringen. Etwas sehr Nützliches ist natürlich das Blinken einer LED ;-). Ein Nucleo-F429ZI Board besitzt drei LEDs. Im Nucleo-Board-Benutzerhandbuch [2] wird in Abschnitt 6.5 gezeigt, an welchen Pins diese LEDs angeschlossen sind:

- PBo: grüne LED
- PB7: blaue LED
- PB14: rote LED

Ändern wir nun die Datei `main.c` und fügen unsere Definitionen für PIN `gpio_set_mode()` hinzu. In der Funktion `main()` setzen wir PB7 mit der blauen LED in den Ausgabemodus und starten eine Endlosschleife. Kopieren wir zunächst die Definitionen für die Pins und GPIO, die wir zuvor besprochen haben. Fügen wir auch zur Bequemlichkeit das Makro `BIT(x)` hinzu - siehe **Listing 4**. Bei einigen Mikrocontrollern werden alle Peripheriegeräte automatisch mit Strom versorgt und aktiviert, wenn sie eingeschaltet werden. Bei STM32-MCUs sind die Peripheriegeräte jedoch standardmäßig deaktiviert, um Strom zu sparen. Um ein GPIO-Peripheriegerät zu aktivieren, muss es über die RCC-Einheit (Reset and Clock Control) aktiviert (getaktet) werden. Im Referenzhandbuch-Abschnitt 7.3.10 erfahren wir, dass das `AHB1ENR` (AHB1 peripheral clock enable register) für das Ein- und Ausschalten von GPIO-Bänken verantwortlich ist. Zunächst fügen wir eine Definition für die gesamte RCC-Einheit hinzu:


```
struct rcc {
    volatile uint32_t CR, PLLCFGR, CFGR, CIR, AHB1RSTR,
    AHB2RSTR, AHB3RSTR, RESERVED0, APB1RSTR, APB2RSTR, RESER-
    VED1[2], AHB1ENR, AHB2ENR, AHB3ENR, RESERVED2, APB1ENR,
    APB2ENR, RESERVED3[2], AHB1LPENR, AHB2LPENR, AHB3LPENR,
    RESERVED4, APB1LPENR, APB2LPENR, RESERVED5[2], BDCR, CSR,
    RESERVED6[2], SSCGR, PLLI2SCFGR;
};
#define RCC ((struct rcc *) 0x40023800)
```

In der Dokumentation des `AHB1ENR`-Registers sehen wir, dass die Bits 0...8 den Takt für die GPIO-Bänke GPIOA-GPIOE einstellen:

```
int main(void) {
    uint16_t led = PIN('B', 7); // Blue LED
    RCC->AHB1ENR |= BIT(PINBANK(led));
    // Enable GPIO clock for LED
    gpio_set_mode(led, GPIO_MODE_OUTPUT);
    // Set blue LED to output mode
    for (;;) asm volatile("nop"); // Infinite loop
    return 0;
}
```

Jetzt müssen wir nur noch herausfinden, wie man einen GPIO-Pin ein- oder ausschaltet, und dann die Hauptschleife modifizieren, um einen LED-Pin einzuschalten, zu verzögern, auszuschalten und zu verzögern. Ein Blick in das Referenzhandbuch, Abschnitt 8.4.7, zeigt, dass das Register `BSRR` für das Setzen der Spannung auf high oder low verantwortlich ist. Die unteren 16 Bits werden verwendet, um das `ODR`-Register zu setzen (das heißt, den Pin auf High zu setzen), und die oberen 16 Bits, um das `ODR`-Register zurückzusetzen (also den Pin auf Low zu setzen). Definieren wir eine API-Funktion für diese Aufgabe:

```
static inline void gpio_write(uint16_t pin, bool val) {
    struct gpio *gpio = GPIO(PINBANK(pin));
    gpio->BSRR |= (1U << PINNO(pin)) << (val ? 0 : 16);
}
```

Als nächstes müssen wir eine Verzögerungsfunktion implementieren. Da wir im Moment keine genaue Verzögerung benötigen, definieren wir eine Funktion `spin()`, die einfach eine bestimmte Anzahl von NOPs ausführt:

```
static inline void spin(volatile uint32_t count) {
    while (count--) asm("nop");
}
```


WEBLINKS

- [1] Referenzhandbuch RM0090 für STM32F429: <https://bit.ly/3neE7S7>
- [2] Benutzerhandbuch Nucleo-144-Board (UM1974): <https://bit.ly/3oIBXKZ>
- [3] Artikel auf GitHub: <https://github.com/cpq/bare-metal-programming-guide>
- [4] Inhalt der Datei link.ld: <https://github.com/cpq/bare-metal-programming-guide/blob/main/step-0-minimal/link.ld>
- [5] Makefile-Tutorial: <https://makefiletutorial.com/>
- [6] Demoprogramm Step 0 minimal : <https://github.com/cpq/bare-metal-programming-guide/blob/main/step-0-minimal>
- [7] Demoprogramm Step 1 blinky: <https://github.com/cpq/bare-metal-programming-guide/blob/main/step-1-blinky>
- [8] .bss (Wikipedia): <https://en.wikipedia.org/wiki/.bss>

Zum Schluss können wir unsere Hauptschleife ändern, um das Blinken der LEDs zu implementieren:

```
for (;;) {
    gpio_write(pin, true);
    spin(999999);
    gpio_write(pin, false);
    spin(999999);
}
```

Der vollständige Quellcode des Projekts befindet sich im Ordner Step 1 blinky [7]. Starten Sie `make flash` und genießen Sie das Blinken der blauen LED!

Im zweiten Teil dieses Artikels werden wir uns die UART-Ausgabe, das Debugging, eine Webserver-Implementierung, automatische Tests und mehr ansehen. Bleiben Sie dran! 

RG – 220665-02

Über den Autor

Sergey Lyubka ist ein Ingenieur und Unternehmer. Er hat einen MSc in Physik von der Staatlichen Universität Kyjiw, Ukraine. Sergey ist Direktor und Mitbegründer von Cesanta, einem Technologieunternehmen mit Sitz in Dublin, Irland (Embedded Web Server for electronic devices: <https://mongoose.ws>). Seine Leidenschaft ist die Programmierung von eingebetteten Bare-Metal-Netzwerken.

Haben Sie Fragen oder Kommentare?

Haben Sie technische Fragen oder Kommentare zu diesem Artikel? Senden Sie eine E-Mail an den Autor unter sergeylyubka@cesanta.com oder kontaktieren Sie Elektor unter redaktion@elektor.de.



Passende Produkte

- > **Dogan Ibrahim, Nucleo Boards Programming with the STM32CubeIDE, Elektor**
<https://elektor.de/19530>
- > **Dogan Ibrahim, Programming with STM32 Nucleo Boards, Elektor**
<https://elektor.de/18585>