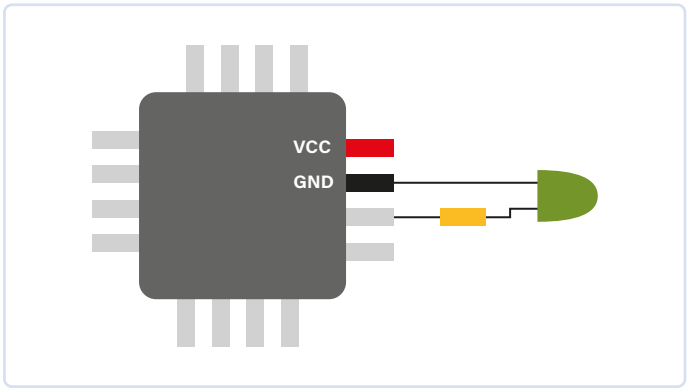
Een gids voor puristisch programmeren (deel 1)

voor STM32 en andere controllers

Sergey Lyubka (Ierland)

Wil je microcontrollers programmeren op een no-nonsense-niveau, waarbij je bovendien inzicht krijgt in hoe ze eigenlijk werken? Deze gids, geschreven voor ontwikkelaars, helpt je op weg met niet meer dan de GCC-compiler en een referentiehandleiding. De principes die je hier leert, helpen je beter te begrijpen hoe frameworks als Cube, Keil en Arduino functioneren. In deze tweedelige gids gebruiken we de STM32F429-controller op een Nucleo-F429ZI-board, maar wat je hier leert kan gemakkelijk worden aangepast aan andere microcontrollers.



Figuur 1. Firmwarecode kan een hoge of lage spanning op een signaalpin zetten, waardoor een LED gaat knipperen.

Een microcontroller (µC, of MCU) is een kleine computer. Doorgaans heeft hij een CPU, RAM, flash-geheugen voor de firmware, en een heleboel aansluitpinnen. Sommige daarvan worden gebruikt om de MCU te voeden (meestal aangeduid met GND (ground) en VCC). Andere pinnen worden gebruikt om met de MCU te communiceren door middel van hoge of lage spanningen die op die pinnen worden gezet. Een van de eenvoudigste communicatiemiddelen is een LED die op een pin is aangesloten: één pootje van de LED is verbonden met de massapin (GND) en de andere via een stroombegrenzende weerstand met een signaalpin. Firmware kan een hoge of lage spanning op die signaalpin zetten, waardoor de LED aan of uit gaat (**figuur 1**).

Benodigde hardware en tools

In deze handleiding gebruiken we een Nucleo-F429ZI-ontwikkelboard (verkrijgbaar bij Mouser en andere leveranciers). Om deze handleiding te volgen, moet je de STM32F429 MCU referentiehandleiding [1] downloaden en vervolgens de handleiding voor het ontwikkelboard [2].

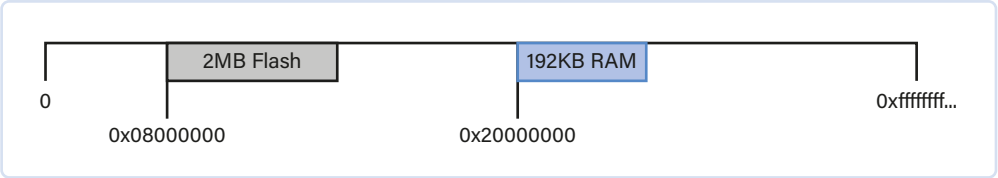
Daarna heb je de volgende tools nodig:
ARM GCC, *https://launchpad.net/gcc-arm-embedded* – voor compileren en linken
GNU make, *https://gnu.org/software/make* – voor build-automatisering
ST link, *https://github.com/stlink-org/stlink* – voor het flashen

Dit zijn de installatie-instructies voor Linux (Ubuntu):

Start een terminal en voer dit uit:

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

Zie [3] om de tools op een Mac of Windows PC te installeren.



Figuur 2. Locatie van STM32F429 flash- en RAM-bereiken.

Geheugen en registers

De adresruimte van een 32bit-MCU, bijvoorbeeld de STM32F429 van STMicroelectronics, is in bereiken verdeeld. Een geheugenbereik (dat op een specifiek adres begint) is bijvoorbeeld toegewezen aan het interne flashgeheugen van de MCU. Instructies in de firmware worden gelezen en uitgevoerd door uit dat bereik te lezen. Een ander bereik is RAM, dat ook op een specifiek adres begint. In het RAM-geheugen kunnen willekeurige waarden worden gelezen en geschreven.

In de STM32F429-referentiehandleiding [1] lezen we in paragraaf 2.3.1 dat het RAM-bereik begint op adres 0x20000000 en een grootte heeft van 192 KB. In paragraaf 2.4 zien we dat het flash-geheugen vanaf adres 0x08000000 is gedefinieerd. Onze MCU heeft 2 MB flash, dus de flash- en RAM-bereiken kunnen zoals in **figuur 2** in kaart worden gebracht.

De handleiding weet ons ook te vertellen dat er nog veel meer geheugenbereiken zijn. De corresponderende adresbereiken zijn gegeven in paragraaf 2.3, 'Memory map'. Er is bijvoorbeeld een 'GPIOA'-bereik dat begint bij 0x40020000 en een lengte heeft van 1 KB.

Deze geheugenbereiken komen overeen met verschillende 'periferie' in de MCU – een plukje siliciumhardware dat ervoor zorgt dat bepaalde pinnen zich op een specifieke manier gedragen. Een perifeer geheugenbereik is een verzameling 32bit-*registers*. Elk register is een bereik van 4 bytes op een bepaald adres in het geheugen, dat overeenkomt met een bepaalde functie van het betreffende randapparaat. Door waarden in een register te schrijven – met andere woorden, door een 32bit-waarde naar een bepaald geheugenadres te schrijven – kunnen we bepalen hoe een bepaald randapparaat zich moet gedragen. Door deze registers uit te lezen, kunnen we de gegevens of configuratie van een randapparaat ophalen/teruglezen.

Er bestaat veel verschillende periferie. Een van de eenvoudigste is GPIO (*General Purpose Input Output*), waarmee de gebruiker MCU-pinnen in 'uitgangsmodus' kan zetten ze een hoge of lage spanning kan laten voeren. Of pinnen in 'ingangsmodus' zetten en er spanningswaarden mee inlezen. Er is een UART-randapparaat dat seriële gegevens kan verzenden en ontvangen via twee pinnen met behulp van een serieel protocol. En er bestaat nog veel meer randapparatuur.

Vaak zijn er meerdere 'instanties' van hetzelfde randapparaat, bijvoorbeeld *GPIOA*, *GPIOB* enzovoort, die verschillende groepen MCU-pinnen aansturen. Evenzo kunnen er *UART1*, *UART2* enzovoort zijn, die de implementatie van meerdere UART-kanalen mogelijk maken. Op de STM32F429 zijn er diverse GPIO- en UART-randapparaten.

De GPIOA-periferie begint bijvoorbeeld op 0x40020000, en we kunnen de beschrijving van het GPIO-register vinden in paragraaf 8.4 [1]. De referentiehandleiding zegt dat het `GPIOA_MODER`-register een offset 0 heeft, wat betekent dat het adres `0x40020000 + 0` is. In **figuur 3** zien we het formaat van het register.

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

pin 15 … pin 8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

pin 7 … pin 3 … pin 2 … pin 1 … pin 0 (A0, B0, …)

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Figuur 3. De beschrijving van de GPIO-registers staat in de referentiehandleiding. Eén MODER-register stuurt 16 fysieke pinnen aan (bron: [1]).

De handleiding laat zien dat het 32-bit *MODER*-register een verzameling van 2bit-waarden is; 16 in totaal. Daarom bestuurt één *MODER*-register 16 fysieke pinnen: bits 0…1 besturen pin 0, bits 2…3 besturen pin 1 enzovoort. De 2bit-waarde codeert de pinmodus: 0 betekent invoer, 1 betekent uitvoer, 2 betekent 'alternatieve functie' – een specifiek gedrag dat elders wordt beschreven – en 3 betekent analoog. Aangezien de naam *GPIOA* is, worden de pinnen 'A0', 'A1' enzovoort genoemd. Voor het randapparaat *GPIOB* zouden de pinnen 'B0', 'B1' … heten.
Als we de 32bit-waarde 0 in het *MODER*-register schrijven, zetten we alle 16 pinnen, van A0 tot en met A15, in de ingangsmodus:

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

Let op de 'qualifier' `volatile`. De betekenis daarvan komt later aan de orde. Door afzonderlijke bits in te stellen, kunnen we selectief specifieke pinnen in een gewenste modus zetten. Dit codefragment bijvoorbeeld zet pin A3 in de uitgangsmodus:

```
* (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
```

Laat me die bitbewerkingen uitleggen. We willen bits 6…7, die verantwoordelijk zijn voor pin 3 van de *GPIOA*-periferie, een bepaalde waarde geven (1 in dit geval). Dat gebeurt in twee stappen. Eerst moeten we de huidige waarde van bits 6…7 wissen, omdat ze mogelijk al een waarde bevatten. Daarna moeten we de relevante bits van 6…7 instellen om de waarde te krijgen die we willen.
Eerst moeten we dus bitbereik 6…7 (twee bits op positie 6) op 0 zetten. Hoe zetten we een aantal bits op 0? De vier stappen zijn in **tabel 1** opgesomd.
Merk op dat de laatste bewerking, een logisch AND-opeatie, N bits op positie X op 0 zet (omdat ze met een 0 worden ge-AND), maar de waarde van alle andere bits behoudt (omdat die met een 1 worden ge-AND). Het behouden van de bestaande waarde is belangrijk, omdat we de instellingen in andere bitbereiken niet willen veranderen. Dus kunnen we in het algemeen, als we N bits op positie X willen wissen, het volgende schrijven:

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

En tenslotte willen we een bepaald bitbereik instellen op de waarde die we willen. We verschuiven die waarde X posities naar links, en OR-ren met de huidige waarde van het hele register (om de waarden van andere bits te behouden):

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

Tabel 1. Specifieke bits 0 maken.

Actie	Uitdrukking	Bits (eerste 12 van 32)
Een getal met N aaneengesloten bits instellen: 2^N-1, hier N=2	`3`	`000000000011`
Verschuif dat getal X posities naar links	`(3<<6)`	`000011000000`
Inverteer het getal: nullen worden enen, en enen worden nullen	`~(3<<6)`	`111100111111`
Logische AND met bestaande waarde	`VAL &= ~(3<<6)`	`xxxx00xxxxxx`

Periferie-programmering die voor mensen leesbaar is

In de vorige paragraaf hebben we gezien dat we een perifeer register kunnen lezen en schrijven door rechtstreeks bepaalde geheugenadressen aan te spreken. Laten we eens kijken naar het codefragment dat pin A3 in uitgangsmodus zet:

```
* (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
```

Dat is tamelijk cryptisch. Zonder uitgebreid commentaar is zulke code moeilijk te begrijpen. We kunnen deze code echter herschrijven in een veel leesbaarder vorm. Het idee is om het hele randapparaat voor te stellen als een structuur met 32bit-velden. Laten we eens kijken welke registers er bestaan voor de GPIO-periferie in paragraaf 8.4 van de referentiehandleiding. Het zijn *MODER*, *OTYPER*, *OSPEEDR*, *PUPDR*, *IDR*, *ODR*, *BSRR*, *LCKR* en *AFR*. Hun offsets zijn 0, 4, 8 enzovoort. Dat betekent dat we ze kunnen voorstellen als een structuur met 32bits-velden, en een #define kunnen opstellen voor *GPIOA*:

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

Dan kunnen we voor het instellen van de GPIO-pinmodus een functie definiëren:

```
// Enum values are per reference manual: 0, 1, 2, 3
enum ;

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
}
```

Nu kunnen we het codefragment voor A3 als volgt herschrijven:

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

Onze MCU heeft diverse GPIO-periferie (ook wel 'banken' genoemd): A, B, C…K. Uit paragraaf 2.3 zien we dat ze 1 KB uit elkaar liggen: *GPIOA* is te vinden op adres 0x40020000, *GPIOB* op 0x40020400,enzovoort:

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

We kunnen een pinnummering definiëren die de bank en het pinnummer omvat. Daarvoor gebruiken we een 2-byte `uint16_t` waarde, waarbij de bovenste byte de GPIO-bank aangeeft en de onderste byte het pinnummer:

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

Zo kunnen we pinnen specificeren voor elke GPIO-bank:

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

Laten we de `gpio_set_mode()`-functie herschrijven op basis van onze pinspecificatie:

```
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
}
```

De code voor A3 spreekt nu voor zich:

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

Merk op dat we zo een nuttige initiële API hebben gemaakt voor de GPIO-periferie. Andere periferie, zoals UART (seriële communicatie) en dergelijke, kan op soortgelijke wijze worden geïmplementeerd. Dit is een goede programmeerpraktijk, omdat het code zelfverklarend en voor mensen leesbaar maakt.

MCU boot- en vectortabel

Wanneer een ARM-MCU opstart, leest hij een zogenaamde 'vectortabel' uit het begin van het flash-geheugen. Een vectortabel is een concept dat alle ARM-MCU's gemeen hebben: een array met 32bit-adressen van interrupt handlers. De eerste 16 ingangen zijn gereserveerd door ARM en zijn gemeenschappelijk voor alle ARM-MCU's. De rest van de interrupt handlers zijn specifiek voor de betreffende MCU – dit zijn interrupt handlers voor periferie. Eenvoudigere MCU's met weinig periferie hebben weinig interrupt handlers en complexere MCU's hebben er veel.
De vectortabel voor de STM32F429 is gedocumenteerd in tabel 62 van de referentiehandleiding [1]. Daaruit blijkt dat er, naast de 16 standaard handlers, 91 perifere handlers zijn.

Elke ingang in de vectortabel is een adres van een functie die de MCU uitvoert wanneer een hardware-interrupt (IRQ) wordt getriggerd. De uitzonderingen zijn de eerste twee ingangen, die een sleutelrol spelen in het opstartproces van de MCU. Deze twee eerste waarden zijn een initiële stackpointer en het adres van de bootfunctie (het beginpunt voor de uitvoering van firmware).
We weten nu dus dat we ervoor moeten zorgen dat onze firmware zo is samengesteld dat de tweede 32bit-waarde in het flash-geheugen het adres van onze bootfunctie bevat. Wanneer de MCU opstart, zal hij dat adres uit het flash-geheugen lezen en naar onze bootfunctie springen.

Minimale firmware

Laten we een bestand maken, *main.c*, en onze bootfunctie specificeren, die aanvankelijk niets doet (in een eindeloze lus terechtkomt), en daarnaast een vectortabel opstellen die 16 standaard-ingangen en 91 STM32-ingangen bevat. Maak en open het bestand *main.c* in een editor naar keuze en tik het volgende in:

```
// 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};
```

Voor de functie `_reset()` hebben we GCC-specifieke attributen `naked` en `noreturn` gebruikt – ze betekenen dat de standaard functie-proloog en -epiloog niet door de compiler moeten worden aangemaakt, en dat de functie niet terugkeert.
De uitdrukking `void (*tab[16 + 91])(void)` betekent: definieer een array van 16 + 91 POINTERS naar functies, die niets retourneren `(void)`, en twee argumenten nemen. Elk van die functies is een IRQ-handler (Interrupt ReQuest-handler). Een array van die handlers wordt een vectortabel genoemd.
De vectortabel `tab` zetten we in een aparte sectie met de naam `.vectors` – die hebben we later nodig om de linker te vertellen dat die sectie helemaal aan het begin van de gegenereerde firmware moet staan, in die volgorde, aan het begin van het flash-geheugen. De eerste twee ingangen zijn de waarde van het stackpointer-register en het beginpunt van de firmware. De rest van de vectortabel laten we gevuld met nullen.

Compileren

Laten we onze code compileren. Start een terminal (of een opdrachtregel in Windows) en voer dit uit:

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

Het werkt! De compilatie leverde een bestand *main.o* op dat onze minimale firmware bevat die niets doet. Het *main.o*-bestand is in ELF binair formaat, dat verschillende secties bevat. We bekijken ze in **listing 1**.

Merk op dat de VMA/LMA sectie-adressen op 0 staan, wat betekent dat *main.o* nog geen complete firmware is omdat het niet de informatie bevat over waar die secties in de adresruimte moeten worden geladen. We moeten een linker gebruiken om het firmwarebestand, *firmware.elf*, te completeren vanuit *main.o*.

De `.text`-sectie bevat firmwarecode, in ons geval slechts een `_reset()`-functie van twee bytes lang – een jump-instructie naar zijn eigen adres. Er is een lege `.data`-sectie en een lege `.bss`-sectie [8] (voor niet-geïnitialiseerde, maar gedeclareerde variabelen wordt deze sectie meestal gevuld met 0). Onze firmware zal worden gekopieerd naar het flash-bereik op offset 0x8000000, maar onze datasectie moet zich in RAM bevinden – daarom moet onze `_reset()`-functie de inhoud van de `.data`-sectie naar RAM kopiëren. Ook moet hij nullen schrijven naar de gehele `.bss`-sectie. In ons geval zijn de `.data`- en `.bss`-secties leeg, maar laten we onze `_reset()`-functie desondanks aanpassen om ze correct te behandelen.

Daartoe moeten we weten waar de stack begint, en waar de data- en bss-secties beginnen. Dit kunnen we aangeven in het *linker*-script, een bestand met instructies aan de linker over waar de verschillende secties in de adresruimte moeten komen, en welke symbolen moeten worden aangemaakt.

Linker-script

Maak een bestand met de naam *link.ld*, en kopieer/plak daar de inhoud uit [4]. Hieronder volgt de uitleg:

```
ENTRY(_reset);
```

Deze regel vertelt de linker de waarde van het 'entry point'-attribuut in de gegenereerde ELF-header – dit is dus een duplicaat van wat een vectortabel heeft. Dit is een hulpmiddel voor een debugger (zoals Ozone, beschreven in het tweede artikel van deze reeks) dat ons helpt een breakpoint in te stellen aan het begin van de firmware. Een debugger kent geen vectortabel, dus vertrouwt hij op de ELF-header.

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

Dit vertelt de linker dat we twee geheugenbereiken in de adresruimte hebben, plus hun adressen en omvang.

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

Dit vertelt de linker om een symbool, `_estack`, aan te maken dat een waarde bevat helemaal aan het einde van het RAM-geheugenbereik. Dit zal onze initiële stackwaarde zijn!

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

Deze regels vertellen de linker de vectortabel eerst in flash te plaatsen, gevolgd door de `.text`-sectie (firmwarecode), gevolgd door alleen-lezen data `.rodata`.

Dan is de `.data`-sectie aan de beurt:

```
.data : {
  _sdata = .; /* .data section start */
  *(.first_data)
  *(.data SORT(.data.*))
```

Listing 1. Compilatie van 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. Opstartcode.

```
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
 }
```

```
  _edata = .; /* .data section end */
} > sram AT > flash
_sidata = LOADADDR(.data);
```

Merk op dat we de linker opdracht geven om `_sdata`- en `_edata`-symbolen aan te maken. We zullen ze gebruiken om de gegevens-sectie naar het RAM te kopiëren in de `_reset()`-functie.
Hetzelfde geldt voor de `.bss`-sectie:

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

Opstartcode

Nu kunnen we onze `_reset()`-functie bijwerken. We kopiëren de `.data`-sectie naar RAM, en initialiseren de `.bss`-sectie met nullen. Dan roepen we de `main()`-functie aan – en komen in een eindeloze lus terecht wanneer `main()` retourneert; zie **listing 2**.
Het diagram van **figuur 4** illustreert hoe `_reset()` `.data` en `.bss` initialiseert.
Het bestand *firmware.bin* is gewoon een aaneenrijging van de drie secties `.vectors` (IRQ-vectortabel), `.text` (code) en `.data` (gegevens). Deze secties zijn opgebouwd volgens het linker-script: `.vectors` staat helemaal aan het begin van het flash-bereik, `.text` volgt onmiddellijk daarna en `.data` ligt ver daarboven. Adressen in `.text` liggen in het flash-geheugenereik, en adressen in `.data` liggen in het RAM-bereik. Als een functie een adres heeft, bijvoorbeeld `0x8000100`, dan bevindt deze zich precies op dat adres in flash. Maar als de code een variabele in de `.data`-sectie benadert met zijn adres, bijvoorbeeld `0x20000200`, dan is er niets op dat adres, omdat bij het opstarten de `.data`-sectie in het bestand firmware.bin zich in flash bevindt! Daarom moet de opstartcode de `.data`-sectie verplaatsen van het flash-geheugenbereik naar het RAM-bereik.
Nu zijn we klaar om het volledige firmware-bestand, *firmware.elf*, te produceren:

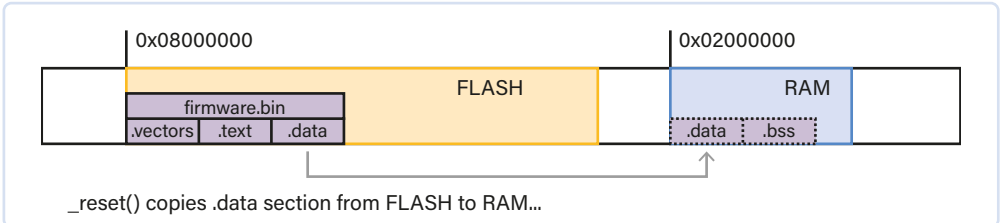
```
$ arm-none-eabi-gcc -T link.ld -nostdlib main.o -o
firmware.elf
```

Laten we secties in firmware.elf nu eens nader bekijken – zie **listing 3**.
Nu zien we dat de `.vectors`-sectie helemaal aan het begin van het flash-geheugen staat, op adres 0x8000000, en de `.text`-sectie direct daarna, op 0x80001ac. Onze code maakt geen variabelen aan, dus er is geen datasectie.

De firmware flashen

We zijn klaar om deze firmware te flashen! Extraheer eerst de secties van *firmware.elf* naar een enkele, aaneengesloten portie binaire code:

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



Figuur 4. De figuur maakt inzichtelijk hoe `_reset()` `.data` en `.bss` initialiseert.

Listing 3. De secties in firmware.elf.

```
$ 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
 ...
```

Gebruik dan het hulpprogramma *st-link* om *firmware.bin* te flashen. Verbind je board via USB en voer dit uit:

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

Klaar! Hiermee heben we firmware geflashed die niets doet.

Makefile: build-automatisering

In plaats van al die compilatie-, link- en flashcommando's in te typen, kunnen we het `make`-hulpprogramma gebruiken om het hele proces te automatiseren. Het `make`-hulpprogramma gebruikt een configuratiebestand met de naam *Makefile*, waarin het instructies leest voor het uitvoeren van acties. Deze automatisering is een geweldige uitvinding omdat het ook het proces van het bouwen van firmware, de gebruikte compilatievlaggen enzovoort documenteert. Er bestaat een heel goede Makefile-tutorial [5]. Degenen die onbekend zijn met `make`, raad ik aan deze door te kijken. Hieronder hebben we de belangrijkste concepten opgesomd die je moet kennen om onze simpele en puristische *Makefile* te begrijpen. Wie al bekend is met `make`, kan dit gedeelte overslaan.

Het *Makefile*-formaat is eenvoudig:

```
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!
```

Nu kunnen we `make` aanroepen met de naam van de actie (ook wel *target* genoemd) om de corresponderende actie uit te voeren:

```
$ make action1
```

Het is mogelijk om variabelen te definiëren en in commando's te gebruiken. Ook kunnen acties de namen zijn van bestanden die moeten worden aangemaakt:

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

Ook kan elke actie een lijst van afhankelijkheden hebben. *firmware.elf* is bijvoorbeeld afhankelijk van ons bronbestand *main.c*. Als het bestand *main.c* verandert, bouwt het `make build`-commando *firmware.elf* opnieuw:

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

Nu zijn we klaar om een *Makefile* te schrijven voor onze firmware. We definiëren een `build` actie/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 definiëren we compilatievlaggen. De `?=` staat voor een standaardwaarde; we kunnen ze op deze manier vanaf de opdrachtregel overschrijven:

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

We specificeren `CFLAGS`-, `LDFLAGS`- en `SOURCES`-variabelen. Dan vertellen we `make`: als je de opdracht krijgt om te bouwen, maak dan een *firmware.elf*-bestand. Het is afhankelijk van het bestand *main.c* en om het te maken moet je de `arm-none-eabi-gcc` compiler starten met de gegeven flags. De speciale variabele `$@` wordt geëxpandeerd tot een target-naam – in ons geval *firmware.elf*.

Laten we `make` aanroepen:

```
$ 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
```

Als we het opnieuw uitvoeren:

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

Listing 4. Blinky LED-sectie van het main.c bestand.

```
#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, OTYPER, 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
}
```

De `make`-utility onderzoekt de modificatietijden voor de *main.c*-dependency en *firmware.elf* – en doet niets als *firmware.elf* actueel is. Maar als we *main.c* veranderen, zal de volgende `make build` opnieuw een compilatie uitvoeren:

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

Wat overblijft is het `flash`-target:

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

Dat is alles! Nu maakt het `make flash`-terminalcommando een *firmware.bin*-bestand aan en flasht het naar het board. Het zal de firmware opnieuw compileren als *main.c* verandert, omdat *firmware.bin* afhankelijk is van *firmware.elf*, en dat is op zijn beurt weer afhankelijk van *main.c*. Dus nu zou de ontwikkelingscyclus uit deze twee acties in een lus bestaan:

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

Het is een goed idee om een schoon target toe te voegen om build-artefacten te verwijderen:

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

De volledige broncode van het project staat in de map *Step 0 minimal* [6].

Knipper-LED

Nu we de hele build-/flash -infrastructuur hebben opgezet, is het tijd om onze firmware te leren iets nuttigs te doen. Zoals het laten knipperen van een LED. Een Nucleo-F429ZI-board heeft drie LED's aan boord. In de geruikershandleiding van het Nucleo-board [2] kunnen we in paragraaf 6.5 zien op welke pinnen deze LED's zijn aangesloten:

- PB0: groene LED
- PB7: blauwe LED
- PB14: rode LED

Laten we het bestand main.c aanpassen en onze definities voor PIN, `gpio_set_mode()` toevoegen. In de `main()`-functie zetten we de blauwe LED op uitgangsmodus en starten we een oneindige lus. Eerst kopiëren we de definities voor de pinnen en GPIO die we eerder hebben besproken. Merk op dat we ook voor het gemak macro toevoegen, `BIT(x)` – zie **listing 4**.
Bij sommige microcontrollers wordt alle periferie automatisch ingeschakeld en van stroom voorzien. Bij STM32-MCU's is de periferie echter standaard uitgeschakeld om energie te sparen. Om een GPIO-randapparaat in te schakelen, moet het expiciet worden ingeschakeld (geklokt) via de RCC-unit (Reset en Clock Control). In paragraaf 7.3.10 van de referentiehandleiding vinden we dat het *AHB1ENR* (AHB1 peripheral clock enable-register) verantwoordelijk is voor het in- of uitschakelen van GPIO-banken. Eerst voegen we een definitie toe voor de complete RCC-unit:

```
struct rcc {
    volatile uint32_t CR, PLLCFGR, CFGR, CIR, AHB1RSTR,
AHB2RSTR, AHB3RSTR, RESERVED0, APB1RSTR, APB2RSTR,
RESERVED1[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 de documentatie van het AHB1ENR-register zien we dat bits 0 tot en met 8 de klok instellen voor GPIO-banken GPIOA...GPIOE:

```
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;
}
```

Nu moeten we nog uitzoeken hoe we een GPIO-pin aan of uit kunnen zetten, en vervolgens de hoofdlus aanpassen om een LED-pin in te schakelen, even tewachten, uit te schakelen en weer even te wachten. Als we een blik werpen op paragraaf 8.4.7 van de referentiehandleiding, zien we dat register *BSRR* verantwoordelijk is voor het hoog of laag maken van een pin. De lage 16 bits worden gebruikt om het *ODR*-register in te stellen (dat wil zegen pin hoog te maken), en de hoge 16 bits worden gebruikt om het *ODR*-register te resetten (dus een pin laag te maken). Laten we daarvoor een API-functie definiëren:

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

Vervolgens moeten we een vertragingsfunctie implementeren. We hebben nu geen nauwkeurige vertraging nodig, dus laten we een functie definiëren met de naam `spin()` die gewoon een bepaald aantal keren een NOP-instructie uitvoert:

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

Nu zijn we klaar om onze hoofdlus aan te passen om een LED te laten knipperen:

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

De volledige broncode van het project is te vinden in de map *Step 1 blinky* [7]. Voer `make flash` uit en geniet van de knipperende blauwe LED!

In het tweede deel van deze reeks komen zaken als UART-uitvoer, debuggen, een webserver-implementatie, automatische tests en meer aan de orde. Tot dan!

220665-03

Over de auteur

Sergey Lyubka is ingenieur en ondernemer. Hij heeft een MSc in natuurkunde van de staatsuniversiteit van Kiev, Oekraïne. Sergey is directeur en medeoprichter van Cesanta, een technologiebedrijf gevestigd in Dublin, Ierland (Embedded Web Server voor elektronische apparaten: https://mongoose.ws). Zijn passie is puristische embedded netwerkprogrammering.

Vragen of opmerkingen?

Hebt u technische vragen of opmerkingen naar aanleiding van dit artikel? Stuur een e-mail naar de auteur via sergey.lyubka@cesanta.com of naar de redactie van Elektor via redactie@elektor.com.

Gerelateerde producten

- **Dogan Ibrahim, *Nucleo Boards Programming with the STM32CubeIDE*, Elektor**
  www.elektor.nl/19530
- **Dogan Ibrahim, *Programming with STM32 Nucleo Boards*, Elektor**
  www.elektor.nl/18585

WEBLINKS

[1] Referentiehandleiding RM0090 voor STM32F429: https://bit.ly/3neE7S7
[2] Nucleo-144 Board gebruikershandleiding (UM1974): https://bit.ly/3oIBXKZ
[3] Dit artikel op GitHub: https://github.com/cpq/bare-metal-programming-guide
[4] Inhoud van het link.ld-bestand: https://github.com/cpq/bare-metal-programming-guide/blob/main/step-0-minimal/link.ld
[5] Makefile-tutorial: https://makefiletutorial.com/
[6] Step 0 minimal demoprogramma: https://github.com/cpq/bare-metal-programming-guide/blob/main/step-0-minimal
[7] Step 1 blinky demoprogramma: https://github.com/cpq/bare-metal-programming-guide/blob/main/step-1-blinky
[8] .bss [Wikipedia]: https://en.wikipedia.org/wiki/.bss