Eine Anleitung zur Bare-Metal-Programmierung

Teil 3: CMSIS-Header, automatische Tests und ein Webserver

Von Sergey Lyubka (Irland)

In den ersten beiden Teilen dieser Artikelreihe haben wir gelernt, wie man Zugriff auf Controller-Pins, den Systemtakt und den UART erhält und unsere ersten Firmware-Beispiele mit Linker-Skripten und Makefiles durchgeführt. In diesem letzten Teil werden uns vordefinierte Header und Bibliotheken das Leben sehr erleichtern. Wir werden einen Webserver programmieren und sehen, wie Builds und Tests einer solchen komplexeren Firmware automatisiert werden können.

Anmerkung der Redaktion: Dieser Leitfaden ist ein „lebendes" Dokument auf GitHub [1].

CMSIS-Header des Herstellers

In den vorherigen Artikeln [2][3] haben wir die Firmware nur mit Hilfe von Datenblättern, einem Editor und einem GCC-Compiler entwickelt. Wir haben die Definitionen der Peripheriestrukturen manuell mit Hilfe von Datenblättern erstellt.
Nun, da Sie wissen, wie das alles funktioniert, ist es an der Zeit, CMSIS-Header einzuführen. Dies sind Header-Dateien mit allen Definitionen, die vom Hersteller der MCU erstellt und bereitgestellt werden. Sie enthalten Definitionen für die internen Blöcke und Peripheriebausteine der jeweiligen MCU und sind daher recht umfangreich.
CMSIS steht für *Common Microcontroller Software Interface Standard* und ist eine gemeinsame Grundlage für MCU-Hersteller, um Peripherie-APIs zu spezifizieren. Da CMSIS ein ARM-Standard ist und die CMSIS-Header vom MCU-Hersteller zur Verfügung gestellt werden, genießt er den Rang einer Definition. Daher ist die Verwendung von Headern der Hersteller der manuellen Erstellung von Definitionen stets vorzuziehen.
Es gibt zwei Arten von CMSIS-Headern:

- ARM Core CMSIS-Header. Sie beschreiben den ARM-Kern und werden von ARM auf GitHub [4] veröffentlicht.
- CMSIS-Header der MCU-Hersteller. Sie beschreiben MCU-Peripherieeinheiten und werden vom MCU-Hersteller veröffentlicht. In unserem Fall veröffentlicht ST sie unter [5].

Wir können diese Header mit einem einfachen Makefile-Snippet abrufen:

(siehe *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-5-cmsis/Makefile*)

```
cmsis_core:
    git clone --depth 1 -b 5.9.0
      https://github.com/ARM-software/CMSIS_5 $@
cmsis_f4:
    git clone --depth 1 -b v2.6.8
      https://github.com/STMicroelectronics/
      cmsis_device_f4 $@
```

Das CMSIS-Paket von ST enthält auch Startdateien für alle MCUs dieses Herstellers. Wir können diese verwenden, anstatt die *startup.c* von Hand zu schreiben. Die von ST bereitgestellte Startup-Datei ruft die Funktion `SystemInit()` auf, also definieren wir sie in *main.c*.
Ersetzen wir nun unsere API-Funktionen in *hal.h* durch CMSIS-Definitionen und lassen wir den Rest der Firmware intakt. Entfernen Sie alle Peripherie-APIs und Definitionen aus *hal.h* und lassen Sie nur die Standard-C-Includes, das CMSIS-Include des Herstellers, die Defines für PIN, BIT, FREQ und die Hilfsfunktion `timer_expired()` übrig.
Wenn wir versuchen, die Firmware neu zu erstellen - `make clean build`, dann wird der GCC sich über fehlende `systick_init()`, `GPIO_MODE_OUTPUT`, `uart_init()` und `UART3` beschweren (und der Vorgang fehlschlagen). Fügen wir diese mit Hilfe der STM32-CMSIS-Dateien hinzu.
Beginnen wir mit `systick_init()`. Die CMSIS-Header des ARM-Kerns bieten eine `SysTick_Config()`-Funktion mit identischer Aufgabe, also verwenden wir diese.
Als nächstes kommt die Funktion `gpio_set_mode()`. Der *stm32f429xx.h*-Header hat eine `GPIO_TypeDef`-Struktur, die mit unserer `struct gpio` identisch ist. Benutzen wir sie:

(siehe *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-5-cmsis/hal.h*)

```
#define GPIO(bank) ((GPIO_TypeDef *)
    (GPIOA_BASE + 0x400U * (bank)))
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) {
  GPIO_TypeDef *gpio =
     GPIO(PINBANK(pin));  // GPIO bank
```

Die Funktionen `gpio_set_af()` und `gpio_write()` sind ebenfalls trivial - ersetzen Sie einfach `struct gpio` durch `GPIO_TypeDef`, und das war's.
Als nächstes kommt UART an die Reihe. Es gibt eine `USART_TypeDef` und Definitionen für USART1, USART2 und USART3. Benutzen wir sie:

```
#define UART1 USART1
#define UART2 USART2
#define UART3 USART3
```

In `uart_init()` und den übrigen UART-Funktionen ändern Sie `struct uart` in `USART_TypeDef`. Der Rest bleibt gleich!
Und schon sind wir fertig. Rebuilden Sie die Firmware und flashen Sie sie erneut. Die LED blinkt, der UART zeigt die Ausgabe an. Herzlichen Glückwunsch, wir haben unseren Firmware-Code so angepasst, dass er die Header-Dateien der Vendor-CMSIS verwendet.
Jetzt sollten wir das Repository ein wenig umorganisieren, indem wir alle Standarddateien in das *include*-Verzeichnis verschieben und das Makefile aktualisieren, damit GCC davon Kenntnis erhält:

(siehe *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-5-cmsis/Makefile*)

```
-I. -Iinclude -Icmsis_core/CMSIS/Core/Include -Icmsis_
f4/Include \
```

Außerdem sollten wir die CMSIS-Header als Abhängigkeit für die Binary einschließen:

```
firmware.elf: cmsis_core cmsis_f4 mcu.h
              link.ld Makefile $(SOURCES)
```

Damit haben wir nun eine Projektvorlage auch für zukünftige Projekte. Den vollständigen Quellcode des Projekts finden Sie im Projektordner *step-5-cmsis* [6].

Einstellen der Takte

Nach dem Booten läuft die Nucleo-F429ZI CPU mit 16 MHz. Die maximale Frequenz beträgt aber 180 MHz. Beachten Sie, dass die Systemtaktfrequenz nicht der einzige Faktor ist, auf den wir achten müssen. Die Peripheriebausteine sind an verschiedene Busse angeschlossen, APB1 und APB2, die unterschiedlich getaktet sind. Ihre Taktgeschwindigkeiten werden durch die Frequenz-Voreinstellungen konfiguriert, die im RCC eingestellt werden (der RCC-Controller verwaltet die Taktfrequenzen für System und Peripherie). Auch die Haupttaktquelle der CPU kann unterschiedlich ausgebildet sein - es kann entweder ein externer Quarzoszillator (HSE) oder ein interner Oszillator (HSI) sein. In unserem Fall wollen wir HSI verwenden.
Wenn die CPU Anweisungen aus dem Flash ausführt, wird die Flash-Lesegeschwindigkeit (die bei etwa 25 MHz liegt) zu einem Engpass, wenn der CPU-Takt höher wird. Es gibt mehrere Tricks, dieses Problem zu umschiffen. Einer davon ist der Befehls-Prefetch. Außerdem können wir dem Flash-Controller einen Hinweis darauf geben, wie schnell der Systemtakt ist: Dieser Wert wird als Flash-Latenzzeit bezeichnet. Bei einem Systemtakt von 180 MHz beträgt der Wert für `FLASH_LATENCY` 5. Bit 8 und Bit 9 im Flash-Controller aktivieren Befehls- und Daten-Caches:

```
FLASH->ACR |= FLASH_LATENCY | BIT(8) |
          BIT(9); // Flash latency, caches
```

Die Taktquelle (HSI oder HSE) durchläuft eine Hardwarekomponente namens PLL, die die Ursprungsfrequenz mit einem bestimmten Wert multipliziert. Dann wird eine Reihe von Frequenzteilern verwendet, um den Systemtakt sowie die APB1- und APB2-Takte einzustellen. Um den maximalen Systemtakt von 180 MHz zu erreichen, sind mehrere Werte von PLL-Teilern und APB-Prescalern möglich. In Abschnitt 6.3.3 im Referenzhandbuch des STM32F4xx-Controllers [7] finden wir die Höchstwerte für den APB1-Takt: ≤ 45 MHz und den APB2-Takt: ≤ 90 MHz. Das beschränkt die Liste der möglichen Kombinationen. Hier haben wir die Werte manuell ausgewählt. Beachten Sie, dass Tools wie *CubeMX* den Prozess automatisieren können und ihn einfach und anschaulich machen.

(siehe *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-6-clock/hal.h*)

```
 // 6.3.3: APB1 clock <= 45MHz;
 //             APB2 clock <= 90MHz
 // 3.5.1, Table 11: configure flash
// latency (WS) in accordance to clock freq
 // 33.4: The AHB clock must be at least
 // 25 MHz when Ethernet is used
 enum { APB1_PRE = 5 /* AHB clock / 4 */,
        APB2_PRE = 4 /* AHB clock / 2 */ };
 enum { PLL_HSI = 16, PLL_M = 8,
        PLL_N = 180, PLL_P = 2 };
               // Run at 180 Mhz
 #define FLASH_LATENCY 5
 #define SYS_FREQUENCY ((PLL_HSI * PLL_N /
                   PLL_M / PLL_P) * 1000000)
 #define APB2_FREQUENCY
         (SYS_FREQUENCY / (BIT(APB2_PRE - 3)))
 #define APB1_FREQUENCY
         (SYS_FREQUENCY / (BIT(APB1_PRE - 3)))
```

Listing 1. Clock-Initialisierung.

(siehe https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-6-clock/sysinit.c)

```
uint32_t SystemCoreClock = SYS_FREQUENCY;

void SystemInit(void) {  // Called automatically by startup code
  SCB->CPACR |= ((3UL << 10 * 2) | (3UL << 11 * 2));  // Enable FPU
  FLASH->ACR |= FLASH_LATENCY | BIT(8) | BIT(9);      // Flash latency, prefetch
  RCC->PLLCFGR &= ~((BIT(17) - 1));                   // Clear PLL multipliers
  RCC->PLLCFGR |= (((PLL_P - 2) / 2) & 3) << 16;      // Set PLL_P
  RCC->PLLCFGR |= PLL_M | (PLL_N << 6);               // Set PLL_M and PLL_N
  RCC->CR |= BIT(24);                                 // Enable PLL
  while ((RCC->CR & BIT(25)) == 0) spin(1);           // Wait until done
  RCC->CFGR = (APB1_PRE << 10) | (APB2_PRE << 13);    // Set prescalers
  RCC->CFGR |= 2;                                     // Set clock source to PLL
  while ((RCC->CFGR & 12) == 0) spin(1);              // Wait until done

  RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;    // Enable SYSCFG
  SysTick_Config(SystemCoreClock / 1000);  // Sys tick every 1ms
}
```

Jetzt sind wir bereit für einen einfachen Algorithmus, um den Takt für CPU und Peripheriebusse einzustellen. Er könnte wie folgt aussehen:

- Optional: FPU einschalten
- Flash-Latenzzeit einstellen
- Entscheiden Sie sich für eine Taktquelle und für die Vorteiler für PLL, APB1 und APB2
- Konfigurieren Sie RCC durch Einstellen der entsprechenden Werte
- Verschieben der Taktinitialisierung in eine separate Datei namens *sysinit.c* mit der Funktion `SystemInit()`, die automatisch vom Startup-Code aufgerufen wird

Siehe **Listing 1**! Wir müssen auch *hal.h* ändern - insbesondere den UART-Initialisierungscode. Die verschiedenen UART-Controller laufen auf unterschiedlichen Bussen: UART1 auf dem schnellen APB2 und die übrigen UARTs auf einem langsameren APB1. Bei einem Standardtakt von 16 MHz spielt das keine Rolle, doch bei höheren Geschwindigkeiten können APB1 und APB2 unterschiedliche Takte haben, so dass wir die Berechnung der Baudrate für den UART anpassen müssen (**Listing 2**).
Nach dem Rebuild und dem erneuten Flashen läuft unser Board mit der maximalen Geschwindigkeit von 180 MHz! Den vollständigen Quellcode des Projekts finden Sie im Projektordner *step-6-clock* [8].

Webserver mit Device-Dashboard

Der Nucleo-F429ZI ist von Haus aus mit Ethernet ausgestattet. Ethernet-Hardware benötigt zwei Komponenten: einen PHY (der elektrische Signale an das Medium, sei es Kupfer oder ein optisches Kabel, sendet/empfängt) und einen MAC (der den PHY-Controller steuert). Auf unserem Nucleo-Board ist der MAC-Controller integriert, und der PHY ist extern (genauer gesagt ist es der LAN8720a von Microchip).

MAC und PHY können über verschiedene Schnittstellen kommunizieren. Wir werden RMII verwenden. Dazu muss eine Reihe von Pins so konfiguriert werden, dass sie ihre alternative Funktion (AF) verwenden. Um einen Webserver zu implementieren, benötigen wir drei Softwarekomponenten:

- einen Netzwerktreiber, der Ethernet-Frames zum/vom MAC-Controller sendet/empfängt
- einen Netzwerk-Stack, der Frames parst und TCP/IP versteht
- eine Netzwerkbibliothek, die HTTP versteht

Wir werden die Netzwerkbibliothek *Mongoose* [9] verwenden, die all dies in einer einzigen Datei implementiert. Mongoose ist doppelt lizenziert (GPLv2/kommerziell) und ermöglicht eine schnelle und einfache Entwicklung von eingebetteten Netzwerken.
Kopieren Sie also *mongoose.c* [10] und *mongoose.h* [11] in unser Projekt. Jetzt haben wir einen Treiber, einen Netzwerk-Stack und eine Bibliothek zur Hand. Mongoose bietet auch viele Beispiele, darunter eines für ein Device-Dashboard [12]. Es enthält viele Dinge wie ein Dashboard-Login, Echtzeit-Datenaustausch über WebSocket, ein eingebettetes Dateisystem, MQTT-Kommunikation und mehr. Lassen Sie uns also dieses schöne Beispiel verwenden. Kopieren Sie zwei zusätzliche Dateien:

- *net.c* [13] - implementiert Dashboard-Funktionalität
- *packed_fs.c* [14] - enthält HTML/CSS/JS GUI-Dateien

Wir müssen Mongoose mitteilen, welche Funktionen aktiviert werden sollen. Das kann über Kompilierungsflags geschehen, die in Präprozessorkonstanten gesetzt werden. Alternativ können die gleichen Konstanten auch in der Datei *mongoose_custom.h* gesetzt werden. Lassen Sie uns der Übersichtlichkeit wegen den zweiten Weg gehen. Erstellen Sie eine Datei *mongoose_custom.h* mit folgendem Inhalt:

Listing 2. UART-Initialisierung.

(siehe https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-6-clock/hal.h)

```
static inline bool uart_init(USART_TypeDef *uart, unsigned long baud) {

  // https://www.st.com/resource/en/datasheet/stm32f429zi.pdf
  uint8_t af = 7;           // Alternate function
  uint16_t rx = 0, tx = 0;  // pins
  uint32_t freq = 0;        // Bus frequency. UART1 is on APB2, rest on APB1

  if (uart == USART1) {
    freq = APB2_FREQUENCY, RCC->APB2ENR |= BIT(4);
    tx = PIN('A', 9), rx = PIN('A', 10);
  } else if (uart == USART2) {
    freq = APB1_FREQUENCY, RCC->APB1ENR |= BIT(17);
    tx = PIN('A', 2), rx = PIN('A', 3);
  } else if (uart == USART3) {
    freq = APB1_FREQUENCY, RCC->APB1ENR |= BIT(18);
    tx = PIN('D', 8), rx = PIN('D', 9);
  } else {
    return false;
  }
```

Listing 3. Ethernet initialisieren, MAC GPIO-Pins aktivieren.

(siehe https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-7-webserver/nucleo-f429zi/main.c)

```
uint16_t pins[] = ;
  for (size_t i = 0; i < sizeof(pins) / sizeof(pins[0]); i++) {
    gpio_init(pins[i], GPIO_MODE_AF, GPIO_OTYPE_PUSH_PULL, GPIO_SPEED_INSANE,
              GPIO_PULL_NONE, 11);
  }
  nvic_enable_irq(61);                          // Setup Ethernet IRQ handler
  RCC->APB2ENR |= BIT(14);                      // Enable SYSCFG
  SYSCFG->PMC |= BIT(23);                       // Use RMII. Goes first!
  RCC->AHB1ENR |= BIT(25) | BIT(26) | BIT(27);  // Enable Ethernet clocks
  RCC->AHB1RSTR |= BIT(25);                     // ETHMAC force reset
  RCC->AHB1RSTR &= ~BIT(25);                    // ETHMAC release reset
```

```
#pragma once
#define MG_ARCH MG_ARCH_NEWLIB
#define MG_ENABLE_MIP 1
#define MG_ENABLE_PACKED_FS 1
#define MG_IO_SIZE 512
#define MG_ENABLE_CUSTOM_MILLIS 1
```

Nun ist es an der Zeit, der Datei *main.c* etwas Netzwerkcode hinzuzufügen. Wir `#include "mongoose.c"`, initialisieren die Ethernet RMII-Pins und aktivieren Ethernet im RCC, siehe **Listing 3**.
Der Mongoose-Treiber verwendet Ethernet-Interrupts, daher müssen wir *startup.c* aktualisieren und *ETH_IRQHandler* zur Vektortabelle hinzufügen. Lassen Sie uns die Definition der Vektortabelle in *startup.c* so umgestalten, dass keine Änderungen erforderlich sind, um eine Interrupt-Handler-Funktion hinzuzufügen. Die Idee ist, ein „weak symbol"-Konzept zu verwenden.

Eine Funktion kann als „weak" gekennzeichnet werden und funktioniert wie eine normale Funktion mit dem Unterschied, dass der Quellcode eine Funktion mit demselben Namen auch an anderer Stelle definiert. Normalerweise führen zwei Funktionen mit demselben Namen dazu, dass ein Build fehlschlägt. Wenn jedoch eine Funktion als „weak" gekennzeichnet ist, gelingt der Build und der Linker wählt eine nicht-schwache Funktion aus. Dies gibt die Möglichkeit, eine „Default"-Funktion in einer Vorlage bereitzustellen, mit der Möglichkeit, sie zu überschreiben, indem einfach eine Funktion mit dem gleichen Namen an anderer Stelle im Code erstellt wird.
So funktioniert es auch in unserem Fall. Wir wollen eine Vektortabelle mit Standard-Handlern füllen, aber dem Benutzer die Möglichkeit geben, jeden Handler zu überschreiben. Zu diesem Zweck erstellen wir eine Funktion `DefaultIRQHandler()` und kennzeichnen sie als `weak`. Dann deklarieren wir für jeden

Listing 4. Initialisierung der Mongoose-Bibliothek.

```
struct mg_mgr mgr;        // Initialise Mongoose event manager
mg_mgr_init(&mgr);        // and attach it to the MIP interface
mg_log_set(MG_LL_DEBUG);  // Set log level
struct mip_driver_stm32 driver_data = {.mdc_cr = 4};  // See driver_stm32.h
struct mip_if mif = {
     .mac {2, 0, 1, 2, 3, 5}
     .use_dhcp = true,
     .driver = &mip_driver_stm32,
     .driver_data = &driver_data,
};
mip_init(&mgr, &mif);
extern void device_dashboard_fn(struct mg_connection *, int, void *, void *);
mg_http_listen(&mgr, "http://0.0.0.0", device_dashboard_fn, &mgr);
MG_INFO(("Init done, starting main loop"));
```

Listing 5. Makefile mit Bibliotheksreferenzen.

```
847 3 mongoose.c:6784:arp_cache_add     ARP cache: added 0xc0a80001 @
90:5c:44:55:19:8b
 84e 2 mongoose.c:6817:onstatechange     READY, IP: 192.168.0.24
 854 2 mongoose.c:6818:onstatechange            GW: 192.168.0.1
 859 2 mongoose.c:6819:onstatechange            Lease: 86363 sec
 LED: 1, tick: 2262
 LED: 0, tick: 2512
```

IRQ-Handler einen Handler-Namen und machen ihn zu einem Alias auf `DefaultIRQHandler()`:

```
void __attribute__((weak)) DefaultIRQHandler(void) {
for (;;) (void) 0;
}
#define WEAK_ALIAS
    __attribute__((weak, alias("DefaultIRQHandler")))
WEAK_ALIAS void NMI_Handler(void);
WEAK_ALIAS void HardFault_Handler(void);
WEAK_ALIAS void MemManage_Handler(void);
...
__attribute__((section(".vectors")))
     void (*tab[16 + 91])(void) =
        { 0, _reset, NMI_Handler,
         HardFault_Handler, MemManage_Handler,
  ...
```

Jetzt können wir einen beliebigen IRQ-Handler in unserem Code definieren, der dann den Standard-Handler ersetzt. Genau das geschieht in unserem Fall: Es gibt eine `ETH_IRQHandler()`-Funktion, die vom STM32-Treiber von Mongoose definiert wurde und die den Standard-Handler ersetzt.
Der nächste Schritt besteht darin, die Mongoose-Bibliothek zu initialisieren: Erstellen Sie einen Event-Manager, richten Sie den Netzwerktreiber ein und starten Sie eine hörende HTTP-Verbindung (**Listing 4**). Jetzt muss nur noch ein `mg_mgr_poll()`-Aufruf in die Hauptschleife eingefügt werden.
Fügen Sie nun die Dateien *mongoose.c*, *net.c* und *packed_fs.c* in das Makefile ein. Führen Sie ein Rebuild durch und flashen Sie das Board erneut. Schließen Sie eine serielle Konsole an die Debug-Ausgabe an und beobachten Sie, dass das Board eine IP-Adresse über DHCP erhält. Siehe **Listing 5**.
Starten Sie einen Browser mit dieser IP-Adresse und erhalten Sie ein funktionierendes Dashboard mit Echtzeitgraphen über WebSocket, MQTT, Authentifizierung und anderen Dingen! In der vollständigen Beschreibung erfahren Sie weitere Details.
Den vollständigen Quellcode des Projekts finden Sie im Verzeichnis *step-7-webserver* [15].

Automatisierte Firmware-Builds (Software CI)

Es ist eine gute Praxis für ein Softwareprojekt, einen kontinuierlichen Integrationstest (CI) durchzuführen. Bei jeder Änderung, die in das Repository übertragen wird, werden alle Komponenten automatisch neu erstellt und getestet.
GitHub gestaltet das ganz einfach. Wir können eine Datei *.github/workflows/test.yml* für die *CI configuration* erstellen. In dieser Datei können wir ARM GCC installieren und `make` in jedem Beispielverzeichnis ausführen, um die entsprechenden Firmware-Dateien zu erstellen. Langer Rede kurzer Sinn, folgendes weist GitHub an, bei jedem Repo-Push zu starten:

(siehe *https://github.com/cpq/bare-metal-programming-guide/blob/main/.github/workflows/test.yml*)

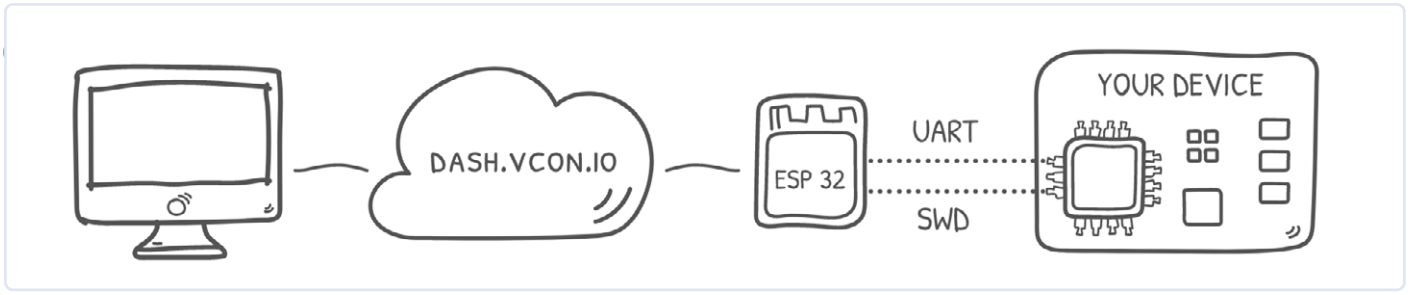

Bild 1. Einrichten des ESP32 als ferngesteuertes Programmiergerät und Registrierung auf vcon.io.

```
name: build
on: [push, pull_request]
```

Dies installiert den ARM-GCC-Compiler:

```
- run: sudo apt -y install
      gcc-arm-none-eabi make stlink-tools
```

Dadurch wird die Firmware in jedem Beispielverzeichnis erstellt:

```
- run: make -C step-0-minimal
- run: make -C step-1-blinky
- run: make -C step-2-systick
- run: make -C step-3-uart
- run: make -C step-4-printf
- run: make -C step-5-cmsis
- run: make -C step-6-clock
- run: make -C step-7-webserver/nucleo-f429zi
- run: make -C step-7-webserver/pico-w
```

Das war's schon! Extrem einfach und extrem leistungsstark. Wenn wir jetzt eine Änderung am Repo vornehmen, die einen Build beschädigt, werden wir von GitHub benachrichtigt. Im Erfolgsfall verhält sich GitHub still. Ein (erfolgreiches) Beispiel dafür zeigt der Lauf in [16].

Automatisierte Firmware-Tests (Hardware CI)

Wäre es nicht toll, die erstellten Firmware-Binärdateien auch auf echter Hardware zu testen, nicht nur, um den Erstellungsprozess zu prüfen, sondern auch, ob die erstellte Firmware korrekt und funktionsfähig ist?
Es ist nicht trivial, ein solches passendes Testsystem zu erstellen. Man kann zum Beispiel eine spezielle Test-Workstation einrichten, ein getestetes Gerät (zum Beispiel ein Nucleo-F429ZI-Board) daran anschließen, eine Software für den Fern-Upload der Firmware schreiben und mit einem eingebauten Debugger testen. Das ist möglich, aber anfällig, erfordert viel Aufwand und viel Aufmerksamkeit.
Die Alternative ist, eines der kommerziellen Hardware-Testsysteme einer *Embedded Board Farm* (EBF) zu verwenden, obwohl solche kommerziellen Lösungen recht teuer sind. Aber es gibt einen genau so einfachen und dennoch preiswerten Weg.

Die Lösung: ESP32 plus vcon.io

Mit Hilfe des Dienstes *http://vcon.io*, der eine Fernaktualisierung der Firmware und einen UART-Monitor implementiert, können wir:

- ein beliebiges ESP32- oder ESP32-C3-Gerät (zum Beispiel ein günstiges Entwicklungsboard) nehmen
- eine vorgefertigte Firmware darauf flashen und den ESP32 in ein ferngesteuertes Programmiergerät verwandeln
- den ESP32 mit dem Zielgerät verbinden: SWD-Pins zum Flashen, UART-Pins zur Erfassung der Ausgabe
- den ESP konfigurieren, um sich auf dem Management-Dashboard von *vcon.io* [17] zu registrieren

Danach verfügt Ihr Zielgerät über eine authentifizierte, sichere RESTful-API zum erneuten Flashen und Erfassen der Geräteausgabe. Sie kann von überall aus aufgerufen werden, zum Beispiel von der Software-KI (siehe **Bild 1**).
Hinweis: vcon.io wird von Cesanta betrieben - dem Unternehmen, für das ich arbeite. Es handelt sich um einen kostenpflichtigen Dienst mit einem Freikontingent: Wenn Sie nur ein paar Geräte zu verwalten haben, ist er völlig kostenlos.

Konfigurieren und Verdrahten des ESP32

Nehmen Sie einen beliebigen ESP32 oder ESP32-C3 - ein Entwicklungsboard, ein Modul oder Ihr eigenes Gerät. Ich empfehle das Entwicklungsboard ESP32-C3 XIAO wegen seines niedrigen Preises und seines kleinen Formfaktors.
Wir gehen davon aus, dass das Zielgerät ein Raspberry Pi W5500-EVB-Pico-Board [18] mit einer eingebauten Ethernet-Schnittstelle ist. Wenn Sie ein anderen Gerät verwenden, passen Sie das Wiring entsprechend seiner Pinbelegung an.

- Folgen Sie *Flashing ESP32* [19], um Ihren ESP32 zu flashen
- Folgen Sie *Network Setup* [20], um ESP32 auf dem Dashboard zu registrieren
- Folgen Sie *Wiring* [21], um ESP32 mit Ihrem Gerät zu verkabeln

Bild 2 zeigt, wie ein konfiguriertes Gerät auf dem Breadboard aussehen kann, und **Bild 3** zeigt, wie es auf dem Dashboard aussieht. Jetzt können Sie Ihr Gerät mit einem einzigen Befehl neu flashen:

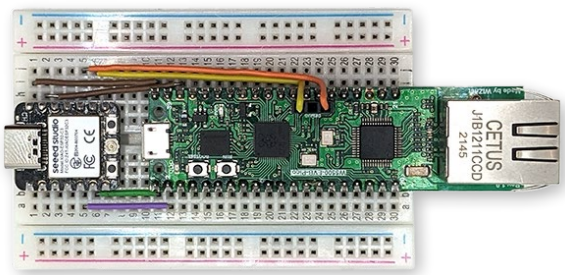

Bild 2. Konfiguriertes Gerät auf dem Breadboard.

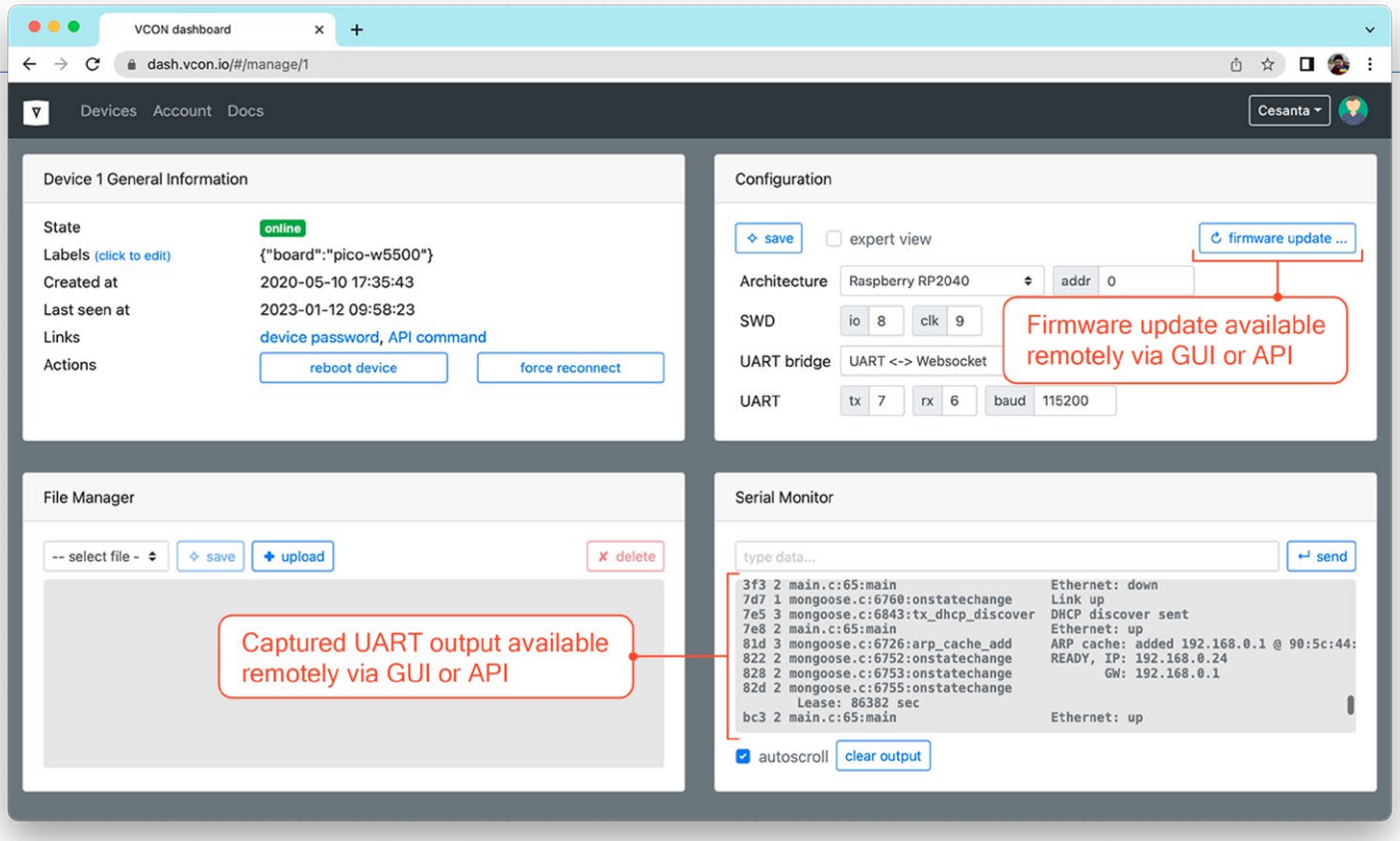

Bild 3. Konfiguriertes Device-Dashboard.

```
curl -su :API_KEY https://dash.vcon.io/api/v3/devices/ID/ota --data-binary @firmware.bin
```

Dabei ist `API_KEY` der Authentifizierungsschlüssel von *dash.vcon.io*, `ID` die registrierte Gerätenummer und `firmware.bin` der Name der neu erstellten Firmware. Sie erhalten den `API_KEY`, indem Sie auf einem Dashboard auf den Link *api key* klicken. Die Geräte-ID ist in der Tabelle aufgeführt. Wir können die Geräteausgabe auch mit einem einzigen Befehl erfassen:

```
curl -su :API_KEY https://dash.vcon.io/api/v3/devices/ID/tx?t=5
```

Dabei bedeutet t=5, dass fünf Sekunden gewartet wird, während die UART-Ausgabe erfasst wird.
Jetzt können wir diese beiden Befehle in jeder Software-CI-Plattform verwenden, um neue Firmware auf einem echten Gerät zu testen und die UART-Ausgabe des Geräts überprüfen.

Integration mit GitHub-Aktionen

Okay, unsere Software-CI erstellt ein Firmware-Image für uns, und jetzt können wir dieses Firmware-Image sogar auf einer echten Hardware testen! Wir sollten ein paar zusätzliche Befehle hinzufügen, die das Utility *curl* verwenden, um die erstellte Firmware an das Testboard zu senden und dann die Debug-Ausgabe zu erfassen. Der *curl*-Befehl erfordert einen geheimen API-Schlüssel, den wir nicht der Öffentlichkeit preisgeben wollen. Im Allgemeinen ist der richtige Weg der folgende:

- Gehen Sie zu den Projekteinstellungen Ihres Projekts auf GitHub (Sie könnten das Repository mit den Webserver-Beispielen klonen), dann auf */Secrets/Actions*
- Klicken Sie auf den Button *New repository secret*
- Geben Sie ihm einen Namen, `VCON_API_KEY`, fügen Sie den Wert in das Feld *Secret* ein und klicken Sie auf *Add secret*

Eines der Beispielprojekte [22] baut Firmware für das RP2040-W5500-Board, also flashen wir es mit einem *curl*-Befehl und dem gespeicherten API-Schlüssel. Der beste Weg ist, ein Makefile-Target zum Testen hinzuzufügen und es von *GitHub Actions* (unserem Software-CI) aufrufen zu lassen:

(siehe *https://github.com/cpq/bare-metal-programming-guide/blob/main/.github/workflows/test.yml*)

```
- run: make -C step-7-webserver/pico-w5500 test VCON_API_KEY=${}
```

Die Umgebungsvariable `VCON_API_KEY` wird an `make` übergeben und Makefile-Ziel von `test` aufgerufen, das unsere Firmware bauen und testen soll. In **Listing 6** sehen Sie das Makefile-Ziel von `test`. Erläuterung:

- Zeile 34: Das `test`-Ziel hängt vom `upload`-Target ab, also wird zuerst `upload` ausgeführt (siehe Zeile 38)
- Zeile 35: Erfasst das UART-Protokoll für fünf Sekunden und speichert es in */tmp/output.txt*
- Zeile 36: Suche nach der Zeichenkette `Ethernet: up` in der Ausgabe und abbrechen, wenn sie nicht gefunden wird
- Zeile 38: Das `upload`-Ziel hängt vom `build` ab, daher wird die Firmware vor dem Testen immer erstellt
- Zeile 39: Wir flashen die Firmware „aus der Ferne“. Das Flag `--fail` bewirkt, dass das Programm *curl* fehlschlägt, wenn die Antwort des Servers nicht erfolgreich ist (nicht HTTP 200 OK ist)

Listing 6. Makefile für Webserver auf dem Pico.

(siehe https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-7-webserver/pico-w5500/Makefile)

```
32 # Requires env variable VCON_API_KEY set
33 DEVICE_URL ?= https://dash.vcon.io/api/v3/devices/1
34 test: update
35   curl --fail -su :$(VCON_API_KEY) $(DEVICE_URL)/tx?t=5 | tee /tmp/output.txt
36   grep 'Ethernet: up' /tmp/output.txt

38 update: build
39   curl --fail -su :$(VCON_API_KEY) $(DEVICE_URL)/ota --data-binary @firmware.bin
```

Listing 7. Ausgabe des Befehls Make Test.

```
$ make test
 curl --fail ...
 {"success":true,"written":59904}
 curl --fail ...
 3f3 2 main.c:65:main                    Ethernet: down
 7d7 1 mongoose.c:6760:onstatechange     Link up
 7e5 3 mongoose.c:6843:tx_dhcp_discover  DHCP discover sent
 7e8 2 main.c:65:main                    Ethernet: up
 81d 3 mongoose.c:6726:arp_cache_add     ARP cache: added 192.168.0.1 @
90:5c:44:55:19:8b
 822 2 mongoose.c:6752:onstatechange     READY, IP: 192.168.0.24
 827 2 mongoose.c:6753:onstatechange           GW: 192.168.0.1
 82d 2 mongoose.c:6755:onstatechange           Lease: 86336 sec
 bc3 2 main.c:65:main                    Ethernet: up
 fab 2 main.c:65:main                    Ethernet: up
```

In **Listing 7** finden Sie die Beispielausgabe des oben beschriebenen Befehls `make test`.

Geschafft! Jetzt stellen unsere automatischen Tests sicher, dass die Firmware erstellt werden kann, dass sie bootfähig ist und dass sie den Netzwerkstack korrekt initialisiert. Dieser Mechanismus kann leicht erweitert werden: Fügen Sie einfach komplexere Aktionen in Ihre Firmware-Binärdatei ein, geben Sie das Ergebnis auf dem UART aus, und überprüfen Sie die erwartete Ausgabe im Test.

Viel Spaß beim Testen! RG– 220665-C-02

Über den Autor

Sergey Lyubka ist Ingenieur und Unternehmer. Er hat einen MSc in Physik von der Staatlichen Universität Kyijw, Ukraine. Sergey ist Direktor und Mitbegründer von Cesanta, einem Technologieunternehmen mit Sitz in Dublin, Irland (Embedded Web Server für elektronische Geräte: https://mongoose.ws). Seine Leidenschaft gilt der Bare-Metal-Programmierung eingebetteter Netzwerke.

Haben Sie Fragen oder Kommentare?

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

WEBLINKS

[1] Diese Anleitung auf GitHub: https://github.com/cpq/bare-metal-programming-guide
[2] Sergey Lyubka, „Anleitung zur Bare-Metal-Programmierung, Teil 1“, Elektor 7-8/2023: https://elektormagazine.de/220665-02
[3] Sergey Lyubka, „Anleitung zur Bare-Metal-Programmierung, Teil 2“, Elektor 9-10/2023: https://elektormagazine.de/220665-B-02
[4] CMSIS, Version 5: https://github.com/ARM-software/CMSIS_5
[5] STM32 CMSIS-Header für die F4-Familie: https://github.com/STMicroelectronics/cmsis_device_f4
[6] Step-5-CMSIS-Ordner: https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-5-cmsis
[7] Referenzhandbuch RM0090 für STM32F4xx Controller (PDF): https://tinyurl.com/stm32f4man
[8] Step-6-CMSIS-Ordner: https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-6-clock
[9] Netzwerkbibliothek Mongoose: https://github.com/cesanta/mongoose
[10] mongoose.c: https://raw.githubusercontent.com/cesanta/mongoose/master/mongoose.c
[11] mongoose.h: https://raw.githubusercontent.com/cesanta/mongoose/master/mongoose.h
[12] Beispielepaket von Mongoose: https://github.com/cesanta/mongoose/tree/master/examples/device-dashboard
[13] Webserver-Beispiel, net.c: https://raw.githubusercontent.com/cesanta/mongoose/master/examples/device-dashboard/net.c
[14] Webserver-Beispiel, packed_fs.c: https://tinyurl.com/packedfsc
[15] Step-7-Webserver-Directory: https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-7-webserver
[16] Beispiel: Erfolgreicher Lauf der Continuous Integration: https://tinyurl.com/bmgoodrun
[17] vcon.io: https://dash.vcon.io/
[18] Raspberry Pi W5500-EVB-Pico-Board: https://docs.wiznet.io/Product/iEthernet/W5500/w5500-evb-pico
[19] Flashing ESP32: https://vcon.io/docs/#module-flashing
[20] Netzwerk-Setup: https://vcon.io/docs/#module-registration
[21] Verdrahtung: https://vcon.io/docs/#module-to-device-wiring
[22] Webserver-Beispiel für Pico: https://tinyurl.com/picowebeg