Eine Anleitung zur Bare-Metal-Programmierung

Teil 2: Exaktes Timing, UART und Debugging

Von Sergey Lyubka (Irland)

Im ersten Teil der Anleitung haben wir gelernt, wie man auf Mikrocontroller-Register zugreift, um Pins zu steuern. Außerdem haben wir eine minimale Firmware und unsere erste Blinkende-LED-Demo mit Hilfe eines Linker-Skripts und eines Makefiles erstellt. In diesem zweiten Teil befassen wir uns mit dem genauen Timing über Systemtakte, dem UART und der Fehlersuche.

Anmerkung des Herausgebers: Diese Anleitung ist ein lebendes und wachsendes Dokument auf GitHub [1]. Daher haben wir uns entschlossen, diesen Artikel um einen weiteren Teil zu erweitern, den Sie in der nächsten Elektor-Ausgabe (11-12/2023) finden werden.

Blinky mit Systick-Interrupt

Für unsere erste LED-Demo „Blinky" im ersten Teil der Anleitung [2] haben wir eine Verzögerungsfunktion namens `spin()` verwendet, die lediglich eine bestimmte Anzahl von NOP-Anweisungen ausgeführt hat. Für eine viel genauere Zeitmessung sollten wir den SysTick-Interrupt von ARM aktivieren. SysTick ist ein 24-Bit-Hardwarezähler, Teil des ARM-Kerns, und daher im Arm-v7-M Architecture Reference Manual [3] dokumentiert ist. Ein Blick in dieses Handbuch zeigt, dass SysTick vier Register hat:

- CTRL - zum Aktivieren/Deaktivieren von SysTick
- LOAD - ein anfänglicher Zählerwert
- VAL - ein aktueller Zählerwert, der bei jedem Taktzyklus dekrementiert wird
- CALIB - Kalibrierungsregister

Jedes Mal, wenn VAL auf Null fällt, wird ein SysTick-Interrupt erzeugt. Der SysTick-Interrupt-Index in der Vektortabelle ist 15, also müssen wir ihn setzen. Beim Booten läuft unser Board Nucleo-F429ZI von STMicroelectronics mit 16 MHz. Wir können den SysTick-Zähler so konfigurieren, dass er jede Millisekunde ein Interrupt auslöst.

Definieren wir zunächst ein SysTick-Peripheral. Wir kennen vier Register, und aus dem Arm-Referenzhandbuch geht hervor, dass die SysTick-Adresse `0xe000e010` ist. Also

```
struct systick {
  volatile uint32_t CTRL, LOAD, VAL, CALIB;
};
#define SYSTICK ((struct systick *) 0xe000e010)
```

Als nächstes fügen wir eine API-Funktion hinzu, die SysTick konfiguriert. Wir müssen den SysTick im `SYSTICK->CTRL`-Register aktivieren und ihn über `RCC->APB2ENR` takten, wie in Abschnitt 7.3.14 des Manuals [4] beschrieben:

```
#define BIT(x) (1UL << (x))
static inline void systick_init(uint32_t ticks) {
  // SysTick timer is 24 bits
  if ((ticks - 1) > 0xffffff) return;
  SYSTICK->LOAD = ticks - 1;
  SYSTICK->VAL = 0;
  // Enable systick
  SYSTICK->CTRL = BIT(0) | BIT(1) | BIT(2);
  RCC->APB2ENR |= BIT(14); // Enable SYSCFG
}
```

Wie bereits erwähnt, läuft das Nucleo-F429ZI-Board mit 16 MHz. Wenn wir `systick_init(16000000 / 1000)` aufrufen, wird also jede Millisekunde ein SysTick-Interrupt erzeugt. Wir sollten eine Interrupt-Handler-Funktion definieren wie folgende, die einfach einen 32-Bit-Millisekunden-Zähler inkrementiert:

```
// "volatile" is important!!
static volatile uint32_t s_ticks;
void SysTick_Handler(void) {
  s_ticks++;
}
```

Bei einem 16-MHz-Takt wird der SysTick-Zähler so initialisiert, dass alle 16.000 Zyklen ein Interrupt ausgelöst wird: Der anfängliche Wert von `SYSTICK->VAL` ist 15.999, dann wird er mit jedem Zyklus

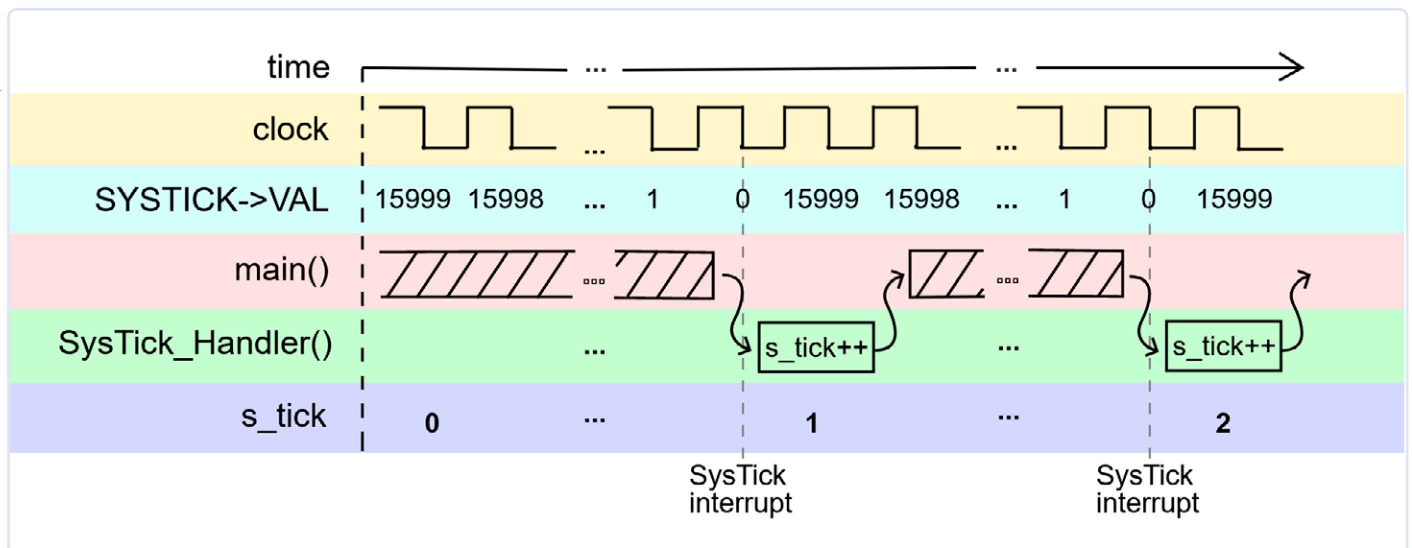

Bild 1. Zeitskalendarstellung der unterbrochenen Firmware-Ausführung mit der Funktion SysTick_Handler().

dekrementiert, bis er 0 erreicht und ein Interrupt ausgelöst wird. Die Ausführung des Firmware-Codes wird unterbrochen und die Funktion `SysTick_Handler()` aufgerufen, um die Variable `s_tick` zu erhöhen. **Bild 1** zeigt, wie dies auf einer Zeitachse aussieht. Der Qualifier `volatile` ist hier erforderlich, weil `s_ticks` vom Interrupt-Handler geändert wird. `volatile` verhindert, dass der Compiler den `s_ticks`-Wert in einem CPU-Register optimiert/zwischenspeichert; stattdessen greift der generierte Code immer auf den Speicher zu. Aus diesem Grund ist der Qualifier `volatile` auch in den peripheren `struct`-Definitionen vorhanden. Es ist wichtig, dies zu verstehen, weshalb wir es anhand der einfachen Arduino-Funktion `delay()` demonstrieren wollen. Verwenden wir unsere Variable `s_ticks`:

```
// This function waits "ms" milliseconds
void delay(unsigned ms) {
  // Time in the future when we need to stop
  uint32_t until = s_ticks + ms; /
  while (s_ticks < until) (void) 0; // Loop until then
}
```

Kompilieren wir nun diesen Code sowohl mit als auch ohne den `volatile`-Qualifier für `s_ticks` und vergleichen wir den kompilierten Assemblercode:

```
// NO VOLATILE: uint32_t s_ticks;
ldr r3, [pc, #8] // cache s_ticks
ldr r3, [r3, #0] // in r3
adds r0, r3, r0 // r0 = r3 + ms
cmp r3, r0 // ALWAYS FALSE
bcc.n 200000d2
bx lr

// VOLATILE: volatile uint32_t s_ticks;
ldr r2, [pc, #12]
ldr r3, [r2, #0] // r3 = s_ticks
adds r3, r3, r0 // r3 = r3 + ms
ldr r1, [r2, #0] // RELOAD: r1 = s_ticks
cmp r1, r3 // compare
bcc.n 200000d2
bx lr
```

Wenn es kein `volatile` gibt, läuft die Funktion `delay()` in einer Endlosschleife und kehrt nie zurück. Das liegt daran, dass sie den Wert von `s_ticks` in einem Register zwischenspeichert (optimiert) und nie aktualisiert. Ein Compiler macht das, weil er nicht weiß, dass `s_ticks` an anderer Stelle durch den Interrupt-Handler aktualisiert wird! Der mit `volatile` kompilierte Code hingegen lädt den `s_ticks`-Wert bei jeder Iteration. Die Faustregel lautet also: **Werte im Speicher, die von Interrupt-Handlern oder von der Hardware aktualisiert werden, müssen als `volatile` deklariert werden!**
Jetzt sollten wir den Interrupt-Handler `SysTick_Handler()` zur Vektortabelle hinzufügen:

```
__attribute__((section(".vectors")))
    void (*tab[16 + 91])(void) = {
    _estack, _reset, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                     0, 0, 0, 0,  SysTick_Handler
};
```

Und damit haben wir schon eine präzise Millisekundenuhr! Lassen Sie uns eine Hilfsfunktion für beliebige periodische Zeitgeber erstellen:

```
// t: expiration time, prd: period,
//    now: current time. Return true if expired
bool timer_expired(uint32_t *t, uint32_t prd,
                   uint32_t now) {
  if (now + prd < *t) *t = 0;
    // Time wrapped? Reset timer
  if (*t == 0) *t = now + prd;
    // First poll? Set expiration
  if (*t > now) return false;
    // Not expired yet,return
  *t = (now - *t) > prd ? now + prd : *t + prd;
    // Next expiration time
  return true; // Expired, return true
}
```

Jetzt können wir unsere Hauptschleife aktualisieren und einen präzisen Timer für das Blinken der LEDs verwenden. In folgendem Beispiel wird Blinkintervall von 250 ms eingestellt:

```
// Declare timer and 500ms period
uint32_t timer, period = 500;
for (;;) {
  if (timer_expired(&timer, period, s_ticks)) {
    static bool on;       // This block is executed
    gpio_write(led, on); // Every "period" milliseconds
    on = !on; // Toggle LED state
  }
  // Here we could perform other activities!
}
```

Beachten Sie, dass wir durch die Verwendung von SysTick mit einer Hilfsfunktion `timer_expired()` unsere Main-Schleife (auch „Superschleife" genannt) nicht blockierend gemacht haben. Das bedeutet, dass wir innerhalb dieser Schleife viele Aktionen durchführen können - zum Beispiel verschiedene Zeitgeber mit unterschiedlichen Zeiträumen, die alle rechtzeitig ausgelöst werden.
Den vollständigen Quellcode des Projekts finden Sie im Ordner *step-2-systick* [5].

UART-Debug-Ausgabe hinzufügen

Nun ist es an der Zeit, unserer Firmware eine für Menschen lesbare Diagnosemöglichkeit hinzuzufügen. Eine der MCU-Peripherien ist eine serielle UART-Schnittstelle. Ein Blick auf die Memory Map in Abschnitt 2.3 des Mikrocontroller-Handbuchs zeigt, dass es mehrere UART/USART-Controller gibt, also Schaltungsteile innerhalb der MCU, die bei entsprechender Konfiguration über bestimmte Pins Daten austauschen können. Eine minimale UART-Konfiguration verwendet die beiden Pins RX (Empfang) und TX (Senden).
In Abschnitt 6.9 des Nucleo-Board-Handbuchs [6] sehen wir, dass der Controller USART3 die Pins PD8 (TX) und PD9 (RX) verwendet und mit dem Debug-Anschluss ST-LINK auf dem Board verbunden ist. Das heißt, wenn wir USART3 konfigurieren und Daten über den PD9-Pin ausgeben, können wir sie über die ST-LINK-USB-Verbindung auf unsere Workstation übertragen.
Lassen Sie uns also eine praktische API für den UART erstellen, so wie wir es für GPIO getan haben. Abschnitt 30.6 [4] fasst die UART-Register zusammen, und hier ist unsere entsprechende UART-`struct`:

```
struct uart {
  volatile uint32_t SR, DR, BRR, CR1, CR2, CR3, GTPR;
};
#define UART1 ((struct uart *) 0x40011000)
#define UART2 ((struct uart *) 0x40004400)
#define UART3 ((struct uart *) 0x40004800)
```

Um einen UART zu konfigurieren, müssen wir:

- den UART-Takt aktivieren, indem das entsprechende Bit in `RCC->APB2ENR` gesetzt wird
- den alternativen Funktions-Pin-Modus für die RX- und TX-Pins einstellen. Je nach verwendetem Peripheriegerät kann es mehrere alternative Funktionen (AF) für einen bestimmten Pin geben. Die AF-Liste finden Sie im STM32F429ZI-Datenblatt, Tabelle 12 [7].
- die Baudrate (Empfangs-/Sendetaktfrequenz) über das BRR-Register einstellen
- den Empfang und das Senden der Peripherie über das CR1-Register aktivieren

Wir wissen bereits, wie man einen GPIO-Pin auf einen bestimmten Modus einstellt. Wenn sich ein Pin im AF-Modus (alternative Funktion) befindet, müssen wir auch die „Funktionsnummer" angeben, also einstellen, welche Peripherie genau die Kontrolle übernimmt. Dies kann über das *Alternate Function Register* `AFR` der GPIO-Peripherie erfolgen. Wenn wir die Beschreibung des AFR-Register im Referenzhandbuch lesen, erfahren wir, dass die AF-Nummer vier Bits belegt, so dass die gesamte Einrichtung für 16 Pins zwei Register belegt.

```
static inline void gpio_set_af(uint16_t pin,
                               uint8_t af_num) {
  struct gpio *gpio = GPIO(PINBANK(pin)); // GPIO bank
  int n = PINNO(pin); // Pin number
  gpio->AFR[n >> 3] &= ~(15UL << ((n & 7) * 4));
  gpio->AFR[n >> 3] |= ((uint32_t) af_num)
                        << ((n & 7) * 4);
}
```

Um den registerspezifischen Code vollständig aus der GPIO-API auszublenden, verschieben wir die GPIO-Takt-Initialisierung in die Funktion `gpio_set_mode()`:

```
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
// Enable GPIO clock
    RCC->AHB1ENR |= BIT(PINBANK(pin));
    ...
```

Nun können wir eine API-Funktion zur UART-Initialisierung erstellen (**Listing 1**).
Schließlich brauchen wir noch Funktionen zum Lesen und Schreiben auf dem UART. Im Referenzhandbuch [4], Abschnitt 30.6.1, erfahren wir, dass das Statusregister SR anzeigt, wenn Daten bereit stehen:

```
static inline int uart_read_ready(struct uart *uart) {
  // If RXNE bit is set, data is ready
  return uart->SR & BIT(5);
}
```

Das Datenbyte selbst kann aus dem Datenregister DR geholt werden:

```
static inline uint8_t uart_read_byte(struct uart *uart) {
  return (uint8_t) (uart->DR & 255);
}
```

Die Übertragung eines einzelnen Bytes kann auch über das Datenregister erfolgen. Nachdem ein Byte zum Schreiben gesetzt wurde,

Listing 1: API-Funktion zur UART-Initialisierung.

```
#define FREQ 16000000  // CPU frequency, 16 Mhz
 static inline void uart_init(struct uart *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

   if (uart == UART1) RCC->APB2ENR |= BIT(4);
   if (uart == UART2) RCC->APB1ENR |= BIT(17);
   if (uart == UART3) RCC->APB1ENR |= BIT(18);
   if (uart == UART1) tx = PIN('A', 9), rx = PIN('A', 10);
   if (uart == UART2) tx = PIN('A', 2), rx = PIN('A', 3);
   if (uart == UART3) tx = PIN('D', 8), rx = PIN('D', 9);

  gpio_set_mode(tx, GPIO_MODE_AF);
   gpio_set_af(tx, af);
   gpio_set_mode(rx, GPIO_MODE_AF);
   gpio_set_af(rx, af);
   uart->CR1 = 0;                           // Disable this UART
   uart->BRR = FREQ / baud;                 // FREQ is a UART bus frequency
   uart->CR1 |= BIT(13) | BIT(2) | BIT(3);  // Set UE, RE, TE
 }
```

muss das Ende der Übertragung abgewartet werden, was über Bit 7 im Statusregister angezeigt wird:

```
static inline void uart_write_byte(struct uart *uart,
                                    uint8_t byte) {
  uart->DR = byte;
  while ((uart->SR & BIT(7)) == 0) spin(1);
}
```

Und das Schreiben eines Puffers:

```
static inline void
  uart_write_buf(struct uart *uart,
                 char *buf, size_t len) {
    while (len-- > 0)
      uart_write_byte(uart, *(uint8_t *) buf++);
}
```

Jetzt initialisieren wir den UART in unserer `main()`-Funktion

```
…
uart_init(UART3, 115200); // Initialize UART
```

und können jedes Mal, wenn die LED blinkt, die Meldung „hi\r\n“ ausgeben!

```
if (timer_expired(&timer, period, s_ticks)) {
  …
  uart_write_buf(UART3, “hi\r\n”, 4); // Write message
}
```

Erstellen Sie den neuen Build, flashen Sie ihn und schließen Sie ein Terminalprogramm an ST-LINK an. Auf meiner Mac-Workstation verwende ich das Programm *cu* (call unix), das natürlich auch unter Linux funktioniert. Für Windows bietet sich das serielle Dienstprogramm PuTTY [8] an. Starten Sie ein Terminal und beobachten Sie die Meldungen:

```
$ cu -l /dev/cu.YOUR_SERIAL_PORT -s 115200
hi
hi
```

Der vollständige Quellcode des Projekts befindet sich im Ordner *step-3-uart* [9].

Umleitung von printf() auf UART

Nun ersetzen wir den `uart_write_buf()`- durch einen `printf()`-Aufruf, der uns die Möglichkeit gibt, Ausgaben zu formatierten, was das Drucken von Diagnoseinformationen flexibler macht. Dazu implementieren wir das sogenannte „printf()-style debugging“.
Die von uns verwendete GNU-ARM-Toolchain enthält nicht nur einen GCC-Compiler und andere Tools, sondern auch eine C-Bibliothek namens *newlib* [10], die von RedHat für eingebettete Systeme entwickelt wurde.
Wenn unsere Firmware eine Standard-C-Bibliotheksfunktion aufruft, zum Beispiel `strcmp()`, dann wird ein *newlib*-Code vom GCC-Linker in unsere Firmware eingefügt.
Einige der Standard-C-Funktionen, insbesondere Operationen für den Datei-Input/Output (IO), werden von *newlib* auf eine besondere Weise implementiert: Diese Funktionen rufen schließlich eine Reihe von Low-Level-IO-Funktionen auf, die als *syscalls* bezeichnet werden. Zum Beispiel:

- `fopen()` ruft schließlich `_open()` auf
- `fread()` ruft schließlich eine untergeordnete Funktion `_read()` auf

Listing 2. Die Funktion main() function wird schön kompakt.

```
#include "hal.h"

static volatile uint32_t s_ticks;
 void SysTick_Handler(void) {
   s_ticks++;
 }

int main(void) {
   uint16_t led = PIN('B', 7);              // Blue LED
   systick_init(16000000 / 1000);           // Tick every 1 ms
   gpio_set_mode(led, GPIO_MODE_OUTPUT);    // Set blue LED to output mode
   uart_init(UART3, 115200);                // Initialise UART
   uint32_t timer = 0, period = 500;        // Declare timer and 500ms period
   for (;;) {
     if (timer_expired(&timer, period, s_ticks)) {
       static bool on;                        // This block is executed
       gpio_write(led, on);                   // Every 'period' milliseconds
       on = !on;                              // Toggle LED state
       uart_write_buf(UART3, "hi\r\n", 4);    // Write message
     }
     // Here we could perform other activities!
   }
   return 0;
 }
```

- `fwrite()`, `fprintf()`, `printf()` rufen schließlich ein Low-Level-`_write()` auf
- `malloc()` ruft schließlich `_sbrk()` auf, und so weiter.

Indem wir einen `_write()`-Syscall modifizieren, können wir `printf()` auf beliebige Ziele umlenken. Dieser Mechanismus wird „IO-Retargeting“ genannt.
Hinweis: Auch die STM32-Cube-IDE verwendet ARM GCC mit *newlib*, weshalb Cube-Projekte typischerweise eine *syscalls.c*-Datei enthalten. Andere Toolchains, zum Beispiel CSS von TI und CC von Keil nutzen möglicherweise eine andere C-Bibliothek mit einem etwas anderen Retargeting-Mechanismus. Wir verwenden *newlib*, also modifizieren wir den `_write()`-Syscall, um auf UART3 zu drucken. Zuvor müssen wir aber unseren Quellcode wie folgt organisieren:

- Verschieben Sie alle API-Definitionen nach *mcu.h*
- Verschieben Sie den Startup-Code nach *startup.c*
- Erstellen Sie eine leere Datei *syscalls.c* für *newlib*-Syscalls
- Ändern Sie das Makefile, um *syscalls.c* und *startup.c* in den Build aufzunehmen.

Nachdem wir alle API-Definitionen nach *mcu.h* verschoben haben, wird unsere Datei *main.c* recht kompakt. Beachten Sie, dass dabei Low-Level-Register nicht auftauchen, sondern nur High-Level-API-Funktionen, die einfach zu verstehen sind - siehe **Listing 2**.
Großartig, jetzt wollen wir `printf()` auf UART3 umleiten. Kopieren Sie in die leere Datei *syscalls.c* den folgenden Code und fügen Sie ihn ein:

```
#include "mcu.h"
int _write(int fd, char *ptr, int len) {
  (void) fd, (void) ptr, (void) len;
  if (fd == 1) uart_write_buf(UART3, ptr, (size_t) len);
  return -1;
}
```

Damit sagen wir: Wenn der Dateideskriptor `fd`, in den wir schreiben, 1 ist (was ein Standardausgabedeskriptor ist), dann schreibe den Puffer in UART3. Andernfalls wird er ignoriert. Das ist die Essenz des Retargeting! Das Rebuilden dieser Firmware führt zu einer Reihe von Linker-Fehlern, wie in **Listing 3** gezeigt. Da wir eine Funktion *newlib stdio* verwendet haben, müssen wir `newlib` mit dem Rest der Syscalls versorgen. Fügen wir nur einen einfachen Stub hinzu, der nichts tut (**Listing 4**).
Jetzt ergibt ein Rebuild keine Fehlermeldung mehr. Im letzten Schritt ersetzen Sie den Aufruf `uart_write_buf()` in der `main()`-Funktion durch einen `printf()`-Aufruf, der etwas Nützliches ausgibt, zum Beispiel einen LED-Status und den aktuellen Wert von Systick:

```
// Write message
printf("LED: %d, tick: %lu\r\n", on, s_ticks);
```

Die serielle Ausgabe sieht wie folgt aus:

```
LED: 1, tick: 250
LED: 0, tick: 500
LED: 1, tick: 750
LED: 0, tick: 1000
```

Herzlichen Glückwunsch! Wir haben erlernt, wie IO-Retargeting funktioniert, und können nun unsere Firmware mit `printf()`

Listing 3. Viele Linker-Fehler.

```
../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/libc_nano.a(lib_a-sbrkr.o): in function `_sbrk_r':
 sbrkr.c:(.text._sbrk_r+0xc): undefined reference to `_sbrk'
 closer.c:(.text._close_r+0xc): undefined reference to `_close'
 lseekr.c:(.text._lseek_r+0x10): undefined reference to `_lseek'
 readr.c:(.text._read_r+0x10): undefined reference to `_read'
 fstatr.c:(.text._fstat_r+0xe): undefined reference to `_fstat'
 isattyr.c:(.text._isatty_r+0xc): undefined reference to `_isatty'
```

Listing 4. Hinzufügen einfacher Stubs.

```
int _fstat(int fd, struct stat *st) {
   (void) fd, (void) st;
   return -1;
 }

void *_sbrk(int incr) {
   (void) incr;
   return NULL;
 }

int _close(int fd) {
   (void) fd;
   return -1;
 }

int _isatty(int fd) {
   (void) fd;
   return 1;
 }

int _read(int fd, char *ptr, int len) {
   (void) fd, (void) ptr, (void) len;
   return -1;
 }

int _lseek(int fd, int ptr, int dir) {
   (void) fd, (void) ptr, (void) dir;
   return 0;
 }
```

debuggen. Den kompletten Quellcode des Projekts finden Sie im Ordner *step-4-printf* [11].

Debuggen mit Segger Ozone

Was ist, wenn unsere Firmware irgendwo feststeckt und `printf()`-Debuggen nicht funktioniert? Was ist, wenn sogar der Startup-Code nicht funktioniert? Dann brauchen wir einen Debugger. Es gibt viele Möglichkeiten, aber ich würde den Debugger Ozone von Segger empfehlen. Warum? Weil er eigenständig ist und keine eingerichtete IDE benötigt. Wir können Ozone direkt mit unserer Datei *firmware.elf* füttern und Ozone wird unsere Quelldateien einsammeln.

Laden Sie also Ozone von der Segger-Website herunter [12]. Bevor wir es mit unserem Nucleo-Board verwenden können, müssen wir nur noch die ST-LINK-Firmware auf dem Onboard-Debugger in die *jlink*-Firmware konvertieren, die Ozone versteht. Folgen Sie dann den Anweisungen auf der Segger-Website [13].

- Starten Sie Ozone. Wählen Sie unser *Device* im Assistenten (**Bild 2**).
- Wählen Sie einen Debugger, den wir verwenden wollen - das sollte ein ST-LINK sein (**Bild 3**).
- Geben Sie den Pfad zu unserer Datei *firmware.elf* an (**Bild 4**).
- Belassen Sie die Standardeinstellungen auf dem nächsten Screen und klicken Sie auf *Finish*. Schon ist unser Debugger geladen (beachten Sie, dass der *mcu.h*-Quellcode übernommen wird), siehe **Bild 5**.
- Klicken Sie auf den grünen Button zum Herunterladen und führen Sie die Firmware aus. Damit sind wir hier fertig (**Bild 6**).

Bild 2. Wählen Sie das Gerät im Assistenten aus.

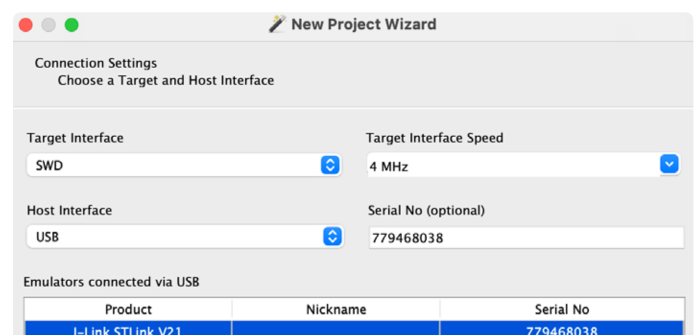

Bild 3. Wählen Sie STLink als Debugger.

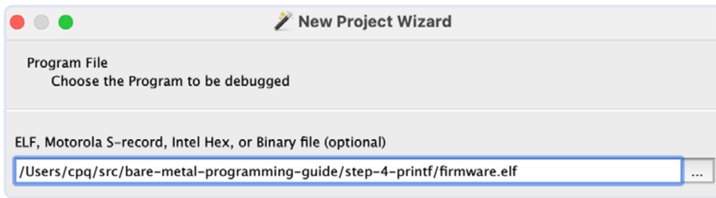

Bild 4. Das zu debuggende Programm ist unsere Datei firmware.elf.

Jetzt können wir in Einzelschritten durch den Code gehen, Haltepunkte setzen und die üblichen Debugging-Maßnahmen durchführen. Besonders hervorzuheben ist die praktische Ozone-Peripherie-Ansicht (**Bild 7**). Mit ihr können wir den Zustand der Peripherieeinheiten direkt untersuchen oder einstellen. Als Beispiel schalten wir die grüne On-Board-LED an PB0 ein:

1. Wir müssen zuerst GPIOB takten. Suchen Sie *Peripherals|RCC|AHB1ENR* und setzen Sie das *GPIOBEN*-Bit (**Bild 8**).
2. Finden Sie *Peripherals|GPIO|GPIOB|MODER* und setzen Sie *MODER0* auf „01" (Ausgang) (**Bild 9**).
3. Suchen Sie *Peripherals|GPIO|GPIOB|ODR* und setzen Sie *ODR0* auf „1" (ein) (**Bild 10**).

Jetzt sollte eine grüne LED leuchten! Viel Spaß beim Debuggen!

Im dritten Teil dieser Artikelreihe werden wir einen Webserver implementieren. Außerdem werden wir zeigen, wie ein Programm automatisch getestet werden kann, und vieles mehr. Bleiben Sie dran!

RG — 220665-B-02

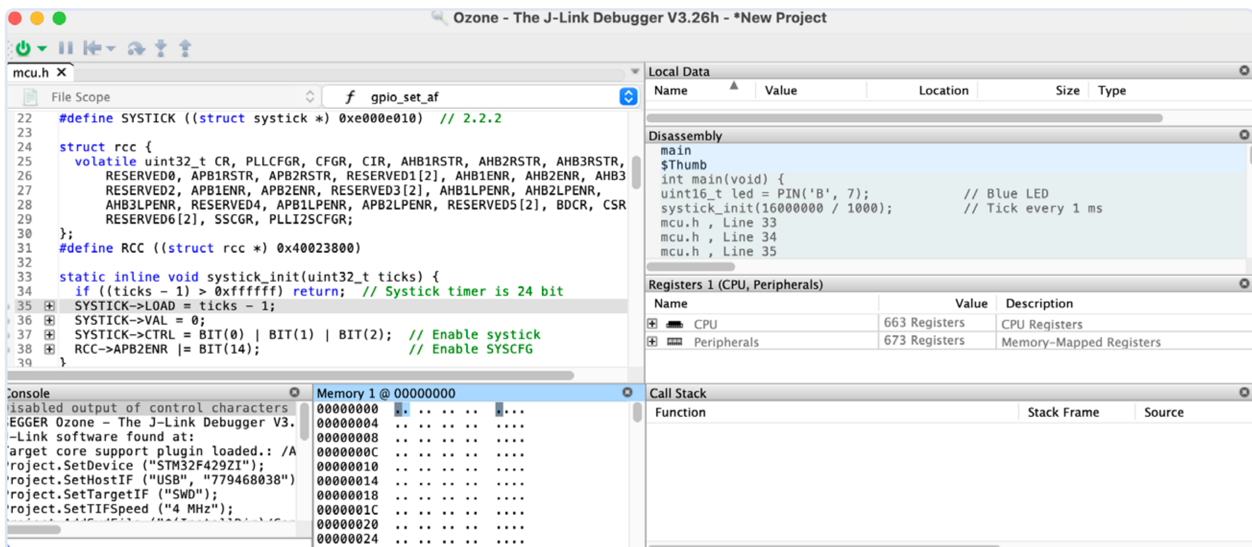

Bild 5. Der Debugger wird geladen, und bald erscheint mcu.h.

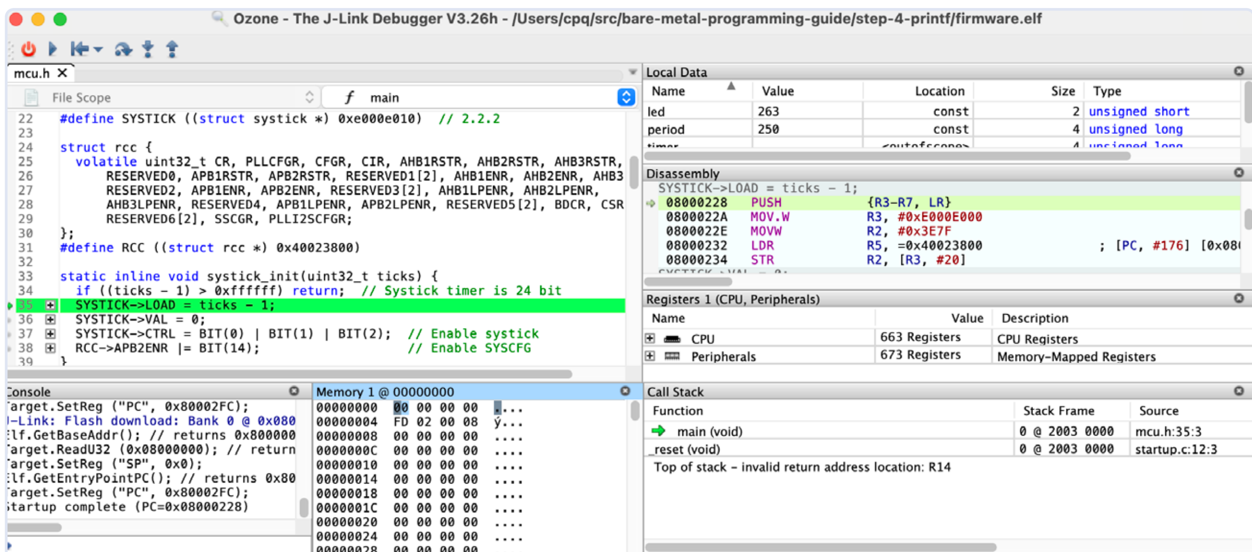

Bild 6. Nachdem wir die Firmware gestartet haben, wird sie in der Zeile SYSTICK->LOAD = ticks - 1; angehalten.

Registers 1 (CPU, Peripherals)

Name	Value	Description
⊞ CPU	663 Registers	CPU Registers
⊞ Peripherals	673 Registers	Memory-Mapped Registers

Bild 7. Praktische Peripherals-Ansicht in Ozone zur einfachen Prüfung und Konfiguration der Peripherals.

Registers 1 (CPU, Peripherals)

Name	Value	Description
⊟ RCC	20 Registers	Reset and clock control
⊞ CR	0000 6E83	clock control register
⊞ PLLCFGR	2400 3010	PLL configuration register
⊞ CFGR		clock configuration register
⊞ CIR		clock interrupt register
⊞ AHB1RSTR		AHB1 peripheral reset register
⊞ AHB2RSTR		AHB2 peripheral reset register
⊞ APB1RSTR	0000 0000	APB1 peripheral reset register
⊞ APB2RSTR	0000 0000	APB2 peripheral reset register
⊟ AHB1ENR	0010 0002	AHB1 peripheral clock register
DMA2EN	0	DMA2 clock enable
DMA1EN	0	DMA1 clock enable
CRCEN	0	CRC clock enable
GPIOHEN	0	IO port H clock enable
GPIOEEN	0	IO port E clock enable
GPIODEN	0	IO port D clock enable
GPIOCEN	0	IO port C clock enable
GPIOBEN	1	IO port B clock enable
GPIOAEN	0	IO port A clock enable

CR – clock control register
Dec 28 291
Hex 0000 6E83
Address 4002 3800

Bild 8. Aktivieren des Takts an Port B durch Setzen des Wertes von GPIOBEN auf 1.

Registers 1 (CPU, Peripherals)

Name	Value	Description
⊞ GPIOH	10 Registers	General-purpose I/Os
⊟ GPIOB	10 Registers	General-purpose I/Os
⊟ MODER	0000 0281	GPIO port mode register
MODER15	b'00	Port x configuration bits (y = 0..15)
MODER14	b'00	Port x configuration bits (y = 0..15)
MODER13	b'00	Port x configuration bits (y = 0..15)
MODER12	b'00	Port x configuration bits (y = 0..15)
MODER11	b'00	Port x configuration bits (y = 0..15)
MODER10	b'00	Port x configuration bits (y = 0..15)
MODER9	b'00	Port x configuration bits (y = 0..15)
MODER8	b'00	Port x configuration bits (y = 0..15)
MODER7	b'00	Port x configuration bits (y = 0..15)
MODER6	b'00	Port x configuration bits (y = 0..15)
MODER5	b'00	Port x configuration bits (y = 0..15)
MODER4	b'10	Port x configuration bits (y = 0..15)
MODER3	b'10	Port x configuration bits (y = 0..15)
MODER2	b'00	Port x configuration bits (y = 0..15)
MODER1	b'00	Port x configuration bits (y = 0..15)
MODER0	b'01	Port x configuration bits (y = 0..15)

Bild 9. Setzen von MODER0 auf 1 (und damit Auswahl des Ausgangs) in den GPIO-Peripherals.

Registers 1 (CPU, Peripherals)

Name	Value	Description
⊞ PUPDR	0000 0100	GPIO port pull-up/pull-down register
⊞ IDR	0000 2199	GPIO port input data register
⊟ ODR	0000 0001	GPIO port output data register
ODR15	0	Port output data (y = 0..15)
ODR14	0	Port output data (y = 0..15)
ODR13	0	Port output data (y = 0..15)
ODR12	0	Port output data (y = 0..15)
ODR11	0	Port output data (y = 0..15)
ODR10	0	Port output data (y = 0..15)
ODR9	0	Port output data (y = 0..15)
ODR8	0	Port output data (y = 0..15)
ODR7	0	Port output data (y = 0..15)
ODR6	0	Port output data (y = 0..15)
ODR5	0	Port output data (y = 0..15)
ODR4	0	Port output data (y = 0..15)
ODR3	0	Port output data (y = 0..15)
ODR2	0	Port output data (y = 0..15)
ODR1	0	Port output data (y = 0..15)
ODR0	1	Port output data (y = 0..15)
⊞ BSRR	0000 0000	GPIO port bit set/reset register

Bild 10. Einschalten von ODR0 durch Auswahl des Werts 1 in ODR (GPIO).

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.

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.

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

WEBLINKS

[1] GitHub-Repository für diesen Artikel: 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] Arm v7-M Architecture Reference Manual: https://developer.arm.com/documentation/ddi0403/ee
[4] Reference Manual RM0090 für STM32F429: https://bit.ly/3neE7S7
[5] Ordner Step 2 SysTick: https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-2-systick
[6] Benutzerhandbuch Nucleo-144 Board (UM1974): https://bit.ly/3oIBXKZ
[7] Datenblatt STM32F429ZI: https://st.com/resource/en/datasheet/stm32f429zi.pdf
[8] PuTTY: https://putty.org
[9] Ordner Step 3 UART: https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-3-uart
[10] C-Bibliothek newlib: https://sourceware.org/newlib
[11] Ordner Step 4 printf: https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-4-printf
[12] Ozone — The J-Link Debugger and Performance Analyzer: https://segger.com/products/development-tools/ozone-j-link-debugger
[13] ST-LINK On-Board in J-Link verwandeln: https://segger.com/products/debug-probes/j-link/models/other-j-links/st-link-on-board