edv-buchversand.de

Embedded Systems

Entwurf und Anwendung von Single-Chip-Systemen mit STM32

» Hier geht's direkt zum Buch

DIE LESEPROBE

Diese Leseprobe wird Ihnen von www.edv-buchversand.de zur Verfügung gestellt.

2.3.3 Connectivity

Warum verfügt das Modul nicht über eine USB-Schnittstelle? In der Tat gibt es eine Vielzahl ähnlicher Breakout-Boards, die gleich einen USB-Anschluss mitbringen. Meist ist das eine Mini-USB-Buchse oder bei neueren Modellen auch schon eine USB-C-Buchse. Das ist in der Tat für schnelle Aufbauten auf einem Steckbrett ideal, weil man sowohl die Datenverbindung als auch die 5-V-Versorgungsspannung vom PC bekommt.

Sobald man das Breakout-Board jedoch auf einer Leiterplatte montieren oder in ein Gehäuse einbauen möchte, wird diese Buchse zum Problem. Man findet viele DIY-Projekte, bei denen Breakout-Boards mit USB unmittelbar an den Rand eines Gehäuses platziert werden mussten, nur damit man die Buchse von außen auch erreichen kann. Eine USB-Buchse direkt auf dem Breakout-Board ist also schon konstruktiv eine unnötige Herausforderung. Abgesehen davon stellt USB immer noch eine recht hohe Hürde bezüglich der Programmierung dar. Unser STM32F412 unterstützt zwar prinzipiell auch USB-OTG (*On-The-Go*), kann also sowohl Device als auch Host sein, das bedeutet aber nicht, dass man beliebige USB-Geräte problemlos anschließen kann.

Letztlich erweisen sich die zwei seriellen UART-Schnittstellen mit 3.3-V-Pegel (RXD0, TXD0 und RXD2, TXD2) dann doch als nützlicher. Man verwendet einfach einen RS232-Pegelwandler oder einen USB-Seriell-Adapter und hat so beide Möglichkeiten, die zudem auch konstruktiv flexibler sind. Entsprechende Buchsen gibt es bereits mit dem erforderlichen Pegelwandler anschlussfertig. Für Prototypen, die man nur auf einem Steckbrett aufbauen möchte, kann man direkt ein USB-Seriell-Kabel verwenden und hat darüber auch gleich die benötigte 5-V-Spannungsversorgung. Der entsprechende Linear-Regler für die 3.3-V-Versorgung des Mikrocontrollers ist auf dem Breakout-Board bereits vorhanden, und für externe Verbraucher stehen die 3.3 V auch zur Verfügung.

Die erste UART, an den Pins TXD0, RXD0, ist primär für Programm-Updates vorgesehen, kann aber sonst für beliebige serielle Geräte verwendet werden, z. B. auch für ein Bluetooth-Modul. Die zweite UART, an den Pins RXD2, TXD2, kann für den Anschluss serieller Sensoren oder eines GPS-Empfängers genutzt werden. Letztlich sind es, bis auf den Programm-Download, zwei gleichwertige UARTs. Darüber hinaus ist es auch möglich, die beiden Pins der zweiten seriellen Schnittstelle (RXD2, TXD2) auch direkt als USB-Schnittstelle oder als CAN-Interface zu verwenden.

Anders als bei den UART-Schnittstellen befindet sich der Pegelwandler für die Ankopplung des Breakout-Boards an ein Modbus-System bereits mit auf dem Board. Die Intention des Breakout-Boards soll unter anderem auch darin bestehen, dass man Sensoren und Aktoren über ein drahtgebundenes Bussystem vernetzen kann (*Sens and Connect*). Als mögliches Bussystem für einfache Anwendungen kommt praktisch nur Modbus

oder ein CAN-Bus in Betracht. Modbus mit seiner *Request-Response-Architektur* ist ein Stück weit einfacher zu handhaben als ein *Multimaster-CAN-Bus*, daher fiel die Wahl auf Modbus. Modbus soll also immer unterstützt werden, während der CAN-Bus optional sein soll. Letztlich können aber beide Bussysteme sogar gleichzeitig genutzt werden, wobei sich für Modbus der Bustreiber bereits mit auf dem Breakout-Board befindet.

2.3.4 Erweiterungen

Eine besonders interessante Erweiterungsmöglichkeit ergibt sich durch die als Systembus genutzte SPI-Schnittstelle. Über den herausgeführten Systembus (SCL, SSI, SSO) kann ein LCD-Grafik-Display mit einer Auflösung von bis zu 240 × 128 Bildpunkten angeschlossen werden. Eine genaue Erklärung dazu findet sich in Abschnitt 4.4, »Grafik-Display«.

Zusätzlich zu dem seriellen LCD-Display kann man noch eine zweite serielle Komponente über die SPI ansteuern. Dafür stehen eine zweite Select-Leitung und ein separates Reset zur Verfügung (/RES, /CS). Denkbar wäre hier etwa ein *W5500-Ethernet-Chip* oder auch ein *RFM95-LoRa-Funktransceiver*. Diese beiden Optionen werden im vorliegenden Buch nicht näher betrachtet. Der Autor hat aber beides in der Praxis bereits erfolgreich implementiert.

Sollte es für eine Anwendung einmal an weiteren digitalen I/O-Pins fehlen, kann man die SPI auch dazu nutzen, Schieberegister als *Port-Expander* anzuschließen. Einfach zu verwendende Schaltkreise, wie SN74HC595 oder SN74HC164, können in dem Fall sehr hilfreich sein.

2.4 Schaltungsentwurf

Schaltungsentwurf ist eine Kunst! In manchen Fällen trifft das sicherlich zu, allerdings ist es zum großen Teil doch eher eine systematische Herangehensweise. Der wesentliche Schritt zu einem funktionierenden Entwurf ist die Aufteilung der Gesamtschaltung in ihre funktionalen Bestandteile (*Dekomposition*). Jeder Mikrocontroller benötigt beispielsweise eine Spannungsversorgung und eine Takterzeugung. Das sind nur zwei von möglicherweise recht vielen Bestandteilen, die man anschließend separat entwirft. Zu jedem Bestandteil gehört wiederum eine minimale Spezifikation. So muss man etwa festlegen, welche Spannung die Versorgung überhaupt liefern soll und welche Stromaufnahme dabei zu erwarten ist. Im einfachsten Fall reicht dafür vielleicht eine überschlägige Rechnung, in schwierigeren Fällen muss man eventuell auch erst eine Simulation durchführen.

Schaltungsentwurf ist ein eigenes Studienfach, und nur mit einem kurzen Abriss von wenigen Zeilen wird niemand zum Meister. Auf der anderen Seite trauen sich viele zu, eine komplexe Software zu erstellen, obwohl sie auch nicht Informatik studiert haben. Der einzige Unterschied besteht darin, dass bei der Schaltungsentwicklung auch einmal etwas nachhaltig kaputtgehen kann, während Programmierer ja immer Strg+Z drücken können.

In den folgenden Abschnitten entwerfen wir nun die Bestandteile der Schaltung unseres Breakout-Boards. Das sind zumeist sehr einfache Schaltungen, was durchaus als Motivation zu verstehen ist, denn schon mit solchen einfachen Schaltungen kommt man recht weit. Den gesamten Schaltplan des Breakout-Boards finden Sie in Anhang A dieses Buchs.

2.4.1 Versorgung

Unser Mikrocontroller hat eine Betriebsspannung von 3.3 V. Üblicherweise hat man jedoch eher eine Versorgungsspannung von 5 V zur Verfügung, etwa wenn das Breakout-Board auf einem Steckbrett direkt über USB versorgt wird. Deshalb sollen die 3.3 V direkt auf dem Breakout-Board generiert werden.

Die einfachste Lösung besteht in dem Fall in der Verwendung eines *Linear-Reglers*. Solche Regler, wie beispielsweise den TS1117BCW33 in unserer Schaltung, gibt es in sehr vielen Varianten. Sie funktionieren alle nach dem gleichen einfachen Prinzip. Ein Transistor wird *linear geregelt*, sodass am Ausgang die gewünschte Spannung U_{Out} anliegt. Am Eingang muss dazu eine Spannung U_{In} anliegen, die mindestens um die Flussspannung des Transistors größer ist. Bei einer Eingangsspannung von $U_{In} = 5V$ und einer Ausgangsspannung von $U_{Out} = 3.3V$ würden folglich $U_{Drop} = U_{In} - U_{Out} = 1.7V$ über dem Transistor abfallen. Dieser Spannungsabfall führt zusammen mit dem Ausgangsstrom I_{Out} zu einer Verlustleistung $P_V = U_{Drop} \cdot I_{Out}$, die der Transistor als Wärme abgeben muss.

Sind Linear-Regler also schlecht? Sie »verheizen« ja förmlich Energie. In der Tat muss die Verlustleistung natürlich gering sein. Das erreicht man entweder, indem man die Spannungsdifferenz U_{Drop} zwischen Ein- und Ausgang sehr klein hält oder indem die Stromaufnahme der nachfolgenden Schaltung sehr gering ist – idealerweise sogar beides. Generell kann man feststellen, dass sich die Linear-Regler gerade als Spannungsversorgung für Mikrocontroller oder auch bei Prozessoren wieder stärker etablieren. Das liegt daran, dass es Varianten gibt, die mit einer sehr geringen *Drop-out-Spannung* auskommen – sogenannte *Low-Drop-Regler* (LDO). Gegenüber schaltenden Reglern erzeugen Linear-Regler nämlich praktisch keine Störungen. Sie bieten darüber hinaus aber noch mehr Vorteile, wie etwa eine schnelle Ausregelung von Schwankungen in der Eingangsspannung.

Solange also die Verlustleistung gering ist, sind Linear-Regler den schaltenden Reglern prinzipiell überlegen. Bei Schaltungen, die mit einer höheren Eingangsspannung betrieben werden, etwa mit 12 V oder gar 24 V, setzt man auf Kombinationen aus einem Schalt- und einem Linear-Regler. Damit reduziert sich die gesamte Verlustleistung auf ein Minimum, und der Mikrocontroller bekommt eine *ripplefreie* Spannungsversorgung. So gesehen ist also der Linear-Regler auf dem Breakout-Board nie überflüssig. Für die Spannungsversorgung unseres Breakout-Boards allein wäre der gewählte Linear-Regler etwas überdimensioniert. Die 3.3 V stehen aber auch an Pin 39 (VDD) nach außen hin zur Verfügung. So können extern angeschlossene Schaltungen, die ebenfalls eine 3.3-V-Versorgung benötigen, direkt darüber betrieben werden. Die Stromaufnahme sollte dabei aber 100 mA nicht übersteigen.

An Pin 2 (VBAT) ist zudem der Anschluss eines *Super-Caps* vorgesehen. Die 3.3 V des Linear-Reglers werden über einen Vorwiderstand und eine Schottky-Diode auf VBAT ausgegeben. Der Vorwiderstand reduziert den Ladestrom beim Einschalten, falls der Super-Cap vollständig entladen ist. Die Schottky-Diode hat gegenüber einer normalen Diode einen geringeren Spannungsabfall von 0.3 V. Damit lädt sich der Super-Cap langsam bis auf 3 V auf. Wenn die Betriebsspannung ausfällt, übernimmt der Super-Cap unterbrechungsfrei die Versorgung der Real Time Clock im Mikrocontroller, sodass zumindest die Uhrzeit korrekt weiterläuft. Super-Caps sind Kondensatoren mit sehr hoher Kapazität. Üblich sind hier Werte von 0.1 bis zu 3.3 F!

Wie lange würde die RTC mit einem Super-Cap als Versorgung denn noch funktionieren? Die Einheit Farad F hat folgende Bedeutung: Bei einem Ampere lädt sich ein Kondensator mit einem Farad in einer Sekunde um ein Volt auf – also: F = As/V. Diese Bedeutung gilt logischerweise auch für den Entladevorgang, weil sonst Ladung anderweitig verloren gehen müsste. Die RTC funktioniert sicher bis zu einer minimalen Spannung von 1.7 V und benötigt, laut Datenblatt des STM32F412, durchschnittlich etwa 1.75 µA. Der Super-Cap hat im voll aufgeladenen Zustand eine Anfangsspannung von 3 V – wir haben demnach einen Spannungshub U_{Delta} von 1.3 V, bis die RTC nicht mehr funktioniert. Mit einem Super-Cap von 1 F würde die RTC also funktionieren:

[2.1] $t_{RTC} = U_{Delta} \cdot C_{Cap} / I_{RTC}$

$t_{RTC} = 1.3\ V \cdot 1\ As/V / 1.75 \cdot 10^{-6}\ A$

$t_{RTC} = 742857\ s$

Das entspricht in etwa 8.9 Tagen. Je nach verwendetem Super-Cap kann man nun mit Gleichung 2.1 die Restlaufzeit der RTC ermitteln.

2.4.2 Reset-Schaltung

Warum benötigt ein Mikrocontroller überhaupt eine Reset-Schaltung? In der Digitalelektronik werden oft sogenannte *Flip-Flops* verwendet. Das sind Logikelemente, die einen 1-Bit-Zustand speichern können. Davon gibt es in einem Mikrocontroller unglaublich viele. Diese Flip-Flops haben allerdings die unschöne Eigenschaft, dass sie nach dem Einschalten zufällig einen der beiden möglichen Zustände annehmen. Das lässt sich nur dadurch korrigieren, dass man zuerst alle Schaltungsteile in einen definierten Anfangszustand bringt. Genau das ist die Aufgabe der Reset-Schaltung.

Jeder Mikrocontroller besitzt dazu einen expliziten Eingang, an dem ein aktiver Pegel, meist der Low-Pegel, den Controller in den Reset-Zustand versetzt. Solange der Pegel aktiv ist, bleibt der Controller im Halt. Damit besteht die einfachste *Reset-Sequenz* darin, diesen Eingang während des Hochlaufens der Versorgungsspannung auf *Low* zu halten. Erst nachdem die Spannung im Mikrocontroller, aber auch an der Außenbeschaltung stabil ist, darf das Reset-Signal den inaktiven High-Pegel annehmen.

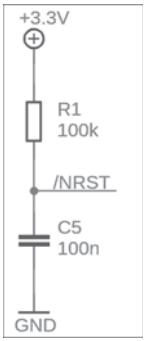

Abbildung 2.3 Reset-Schaltung in RC-Kombination (Tiefpass)

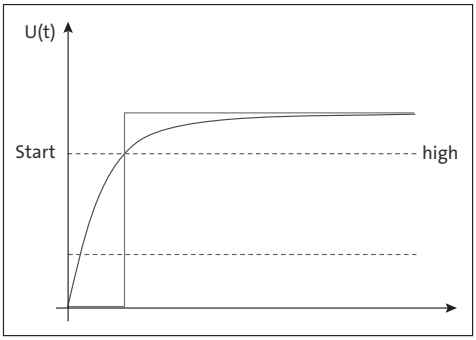

Abbildung 2.4 Spannungsverlauf der Reset-Schaltung an /NRST

Schaltungstechnisch lässt sich das einfach durch die obige RC-Kombination erreichen. Der Kondensator ist zu Beginn nicht aufgeladen und zieht einen hohen Ladestrom. Damit stellt er quasi eine Verbindung nach Masse dar, was den Pegel an /NRST kurzzeitig auf *Low* zieht. Durch den Widerstand lädt sich der Kondensator langsam auf – die Spannung an /NRST nähert sich dadurch exponentiell der Betriebsspannung. Der Reset-Eingang hat zudem *Schmitt-Trigger-Verhalten* und kippt ab einem konkreten Spannungswert direkt in den inaktiven Zustand.

Wird die Betriebsspannung abgeschaltet, entlädt sich der Kondensator über den Widerstand bzw. auch über die interne Schutzdiode des /NRST-Eingangs.

Der STM32F412 hat intern bereits einen *Pull-up-Widerstand* von 40 bis 50 kOhm am /NRST-Eingang. So gesehen könnte man den Widerstand außen theoretisch sogar weglassen. Durch die Parallelschaltung wird der resultierende Pull-up-Widerstand jedoch etwas niederohmiger, was für stabilere Verhältnisse an diesem wichtigen Eingangspin sorgt.

Die hier angegebenen Werte halten den Reset-Eingang etwa 2 ms aktiv.

2.4.3 Takterzeugung

Die Takterzeugung wird bei Mikrocontrollern oft mittels *Pierce-Oszillator* realisiert. Dazu befindet sich im Mikrocontroller ein *invertierender Verstärker*, dessen Ein- und Ausgang auf zwei Pins geführt sind. Als Außenbeschaltung wird dann ein *Bandpassfilter*, bestehend aus einem Quarz mit zwei Kondensatoren, benötigt. Aufgrund des Aussehens dieses Filters wird es auch *Pi-Filter* genannt. Der Quarz ist das frequenzbestimmende Element und sorgt für ein sehr schmalbandiges Filter. Die beiden wichtigsten Eigenschaften dieses Filters sind eine sehr hohe Güte bei der Resonanzfrequenz sowie eine Phasendrehung um 180°. Der invertierende Verstärker im Inneren und das Filter außen haben dann zusammen eine Phasendrehung von 360°, was zu einer *Mitkopplung* und damit zu einer Schwingneigung führt.

Die beiden Lastkapazitäten im Bereich von 12 bis 33 pF sind zwar klein, aber trotzdem wichtig. Je kleiner ihr Wert gewählt wird, desto größer wird die Verstärkung, und desto sicherer schwingt der Oszillator an. Zu klein oder zu groß dürfen sie jedoch nicht sein, weil dann die 180°-Phasenbedingung kippen könnte und der Oszillator gar nicht mehr schwingt. Außerdem beeinflussen sie geringfügig die exakte Frequenz des Oszillators, weshalb sie auch »Ziehkondensatoren« genannt werden. Das ist insbesondere beim 32.768-kHz-Uhrenquarz für die RTC relevant. Die Uhr kann man softwareseitig noch feintrimmen, aber dann sollten sich die Werte der Kondensatoren außen natürlich auch nicht mehr ändern.

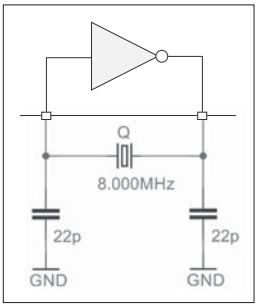

Abbildung 2.5 Pierce-Oszillator, invertierender Verstärker mit Pi-Filter

Als Lastkapazitäten sollten daher immer *Vielschicht-Keramik-Kondensatoren* verwendet werden, weil diese eine sehr geringe Temperaturabhängigkeit haben. Die Temperaturabhängigkeit ist natürlich auch ein Faktor bei der Wahl des Quarzes. In der Regel wird der Wert in ppm/C^2 (Parts per Million pro Grad Celsius zum Quadrat) angegeben und sollte um 0.035 ppm/C^2 liegen.

2.4.4 Digitale I/O-Pins

Prinzipiell kann jeder Pin unseres Mikrocontrollers entweder als Eingang (*Input*) oder als Ausgang (*Output*) konfiguriert werden. Bei einem Eingang kann man zudem noch einen *Pull-up*-Widerstand oder alternativ einen *Pull-down*-Widerstand aktivieren, und bei einem Ausgangspin gibt es die Wahl zwischen einem *Push-Pull*- und einem *Open-Drain*-Treiber. Es sind sogar *bidirektionale* Pins möglich, indem man einen Open-Drain-Ausgang mit einem Pull-up-Widerstand kombiniert. Das ist sehr flexibel, und daher käme es gar nicht darauf an, welchen Pin wir bei unserem Breakout-Board an die Steckerleisten führen – man könnte einfach das nehmen, was am nächsten liegt.

Die Auswahl wird aber etwas komplizierter, wenn man statt eines einfachen I/O eine der vielen Sonderfunktionen (*Alternate Functions*) des STM32 nutzen möchte. Im Datenblatt des STM32F412, aber auch jedes anderen Typs, findet man dazu mehrere Tabellen, jeweils eine pro Port, mit allen möglichen Alternate Functions (AF) zu jedem einzelnen Pin. Theoretisch hat darin jeder Pin 16 mögliche Sonderfunktionen, wobei einige aber nicht belegt sind. Die nachfolgende Tabelle ist ein Auszug aus dem Datenblatt des STM32F412 für Port B.

In der linken Spalte der Tabelle findet man alle Pins von Port B und in den Spalten daneben die möglichen Sonderfunktionen AF0 bis AF15. Betrachtet man exemplarisch die Zeile für Pin 6 (PB6), sieht man, dass dieser Pin etwa mit dem ersten Kanal von Timer 4 verbunden werden kann (AF2: TIM4_CH1) oder als Taktleitung für I^2C 1 dienen kann (AF4:

I2C1_SCL) oder auch als TX-Leitung der UART 1 genutzt werden kann. Viele der Sonderfunktionen kommen dabei mehrfach vor – entweder in derselben Tabelle oder in einer der anderen Tabellen. Damit hat man zwar viele Möglichkeiten, allerdings gleicht die optimale Auswahl schon einer Sisyphusarbeit. In der Praxis hat sich diese Matrix jedoch mehrfach als hilfreich erwiesen, weil man durch geschickte Zuordnung auch das Routing einer Leiterplatte vereinfachen kann, indem man dieselbe Funktion einem anderen Pin zuweist, der günstiger zu routen ist.

		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF12	AF15
Port B	PB0	-	TIM1_CH2N	TIM3_CH3	TIM8_CH2N	-	-	SPI5_SCK/ I2S5_CK	-	-	-	-	-	EVENTOUT
	PB1	-	TIM1_CH3N	TIM3_CH4	TIM8_CH3N	-	-	SPI5_NSS/ I2S5_WS	-	DFSDM1_ DATIN0	QUADSPI_ CLK	-	-	EVENTOUT
	PB2	-	-	-	-	-	-	DFSDM1_ CKIN0	-	-	QUADSPI_ CLK	-	-	EVENTOUT
	PB3	JTDO-SWO	TIM2_CH2			I2CFMP1_ SDA	SPI1_SCK/I2S1_CK	SPI3_SCK/ I2S3_CK	USART1_RX	-	I2C2_SDA	-	-	EVENTOUT
	PB4	JTRST	-	TIM3_CH1	-	-	SPI1_MISO	SPI3_MISO	I2S3ext_ SD	-	I2C3_SDA	-	SDIO_D0	EVENTOUT
	PB5	-	-	TIM3_CH2	-	I2C1_SMBA	SPI1_MOSI/I2S1_SD	SPI3_MOSI/ I2S3_SD	-	-	CAN2_RX	-	SDIO_D3	EVENTOUT
	PB6	-	-	TIM4_CH1	-	I2C1_SCL	-	-	USART1_TX	-	CAN2_TX	QUADSPI_ BK1_NCS	SDIO_D0	EVENTOUT
	PB7	-	-	TIM4_CH2	-	I2C1_SDA	-	-	USART1_RX	-	-	-	FSMC_NL	EVENTOUT
	PB8	-	-	TIM4_CH3	TIM10_CH1	I2C1_SCL	-	SPI5_MOSI/I2S5_SD	-	CAN1_RX	I2C3_SDA	-	SDIO_D4	EVENTOUT
	PB9	-	-	TIM4_CH4	TIM11_CH1	I2C1_SDA	SPI2_NSS/ I2S2_WS	-	-	CAN1_TX	I2C2_SDA	-	SDIO_D5	EVENTOUT
	PB10	-	TIM2_CH3	-	-	I2C2_SCL	SPI2_SCK/ I2S2_CK	I2S3_MCK	USART3_TX	-	I2CFMP1_ SCL	-	SDIO_D7	EVENTOUT
	PB11	-	TIM2_CH4	-	-	I2C2_SDA	I2S2_CKIN	-	USART3_RX	-	-	-	-	EVENTOUT
	PB12	-	TIM1_BKIN	-	-	I2C2_SMBA	SPI2_NSS/ I2S2_WS	SPI4_NSS/ I2S4_WS	SPI3_SCK/ I2S3_CK	USART3_ CK	CAN2_RX	DFSDM1_ DATIN1	FSMC_D13/FSMC_DA13	EVENTOUT
	PB13	-	TIM1_CH1N	-	-	I2CFMP1_ SMBA	SPI2_SCK/ I2S2_CK	SPI4_SCK/ I2S4_CK	-	USART3_ CTS	CAN2_TX	DFSDM1_ CKIN1	-	EVENTOUT
	PB14	-	TIM1_CH2N	-	TIM8_CH2N	I2CFMP1_ SDA	SPI2_MISO	I2S2ext_SD	USART3_ RTS	DFSDM1_ DATIN2	TIM12_CH1	FSMC_D0	SDIO_D6	EVENTOUT
	PB15	RTC_ 50Hz	TIM1_CH3N	-	TIM8_CH3N	I2CFMP1_ SCL	SPI2_MOSI/I2S2_SD	-	-	DFSDM1_ CKIN2	TIM12_CH2	-	SDIO_CK	EVENTOUT

Abbildung 2.6 Alternate Functions an Port B (Datenblatt STM32F412)

Machen wir es konkret. Unser Breakout-Board soll sechs Ausgänge haben, die alle in der Lage sind, PWM-Signale zu generieren. In dem Fall dürfen wir also nur Pins verwenden, die mit einem Timer-Kanal verbunden werden können. Bei Port B kann man fast jeden Pin mit einem Timer verbinden, aber PB2 dürften wir gerade nicht verwenden. Stattdessen nehmen wir PB4 bis PB9 als OUT0 bis OUT5. Diese Pins liegen, was das Layout angeht, alle recht günstig zur vorgesehenen Steckerleiste unseres Boards.

Auf diese Weise legt man zuerst alle Pins fest, die eine Sonderfunktion haben sollen – etwa die UARTs und die SPI. Das betrifft dann auch die Eingänge. Wenn man beispielsweise Impulsbreiten mittels Timer messen möchte, muss der entsprechende Eingangspin auch auf einen Timer-Kanal verbunden werden können. Die Eingänge IN0 bis IN3 entsprechen bei unserem Breakout-Board den Pins PC6 bis PC9 am Mikrocontroller, und

diese Pins können wahlweise auf Timer 3 oder Timer 8 verbunden werden. Damit besteht also auch bei den Eingängen die Möglichkeit, Timer-Funktionen zu nutzen.

Nutzung der Port-Tabellen

Versuchen Sie bitte einmal, mithilfe der Port-Tabellen im Datenblatt des STM32F412 und anhand der Schaltung des Breakout-Boards in Anhang A nachzuvollziehen, wie man die Eingänge IN0 bis IN3 auf die Kanäle von Timer 3 und gleichzeitig OUT0 bis OUT3 auf die Kanäle von Timer 4 verbinden kann.

Alle anderen Pins, für die eine einfache I/O-Funktion ausreicht, können dann wahlfrei oder entsprechend einem günstigeren Layout zugeordnet werden. Solche Pins sind etwa die Ausgänge für die LEDs oder die Select-Leitung für den EEPROM.

Mitunter benötigt man mehrere Iterationen, bis man die optimale Zuordnung aller Pins in einer Schaltung gefunden hat.

2.4.5 Externe Interrupts

Eine besondere Gruppe von Eingängen sind die, die unmittelbar zu einer Unterbrechung (*External Interrupt*) des Programms führen. Mit den Alternate Functions gibt es ja bereits die Möglichkeit, Sonderfunktionen der Hardware zu nutzen – hier ist jedoch eine universelle Funktionalität gemeint, die man frei programmieren kann. Ein externer Interrupt ruft im Grunde eine Funktion in der Software auf, was sehr flexibel ist.

Unser Breakout-Board soll über zwei solcher *interruptfähigen* Eingänge INT0 und INT1 verfügen. Die Frage ist, welche Pins des Mikrocontrollers man dafür nimmt. Beim STM32F4 kann man praktisch jeden Pin so konfigurieren, dass es bei einer Signalflanke einen Interrupt auslöst, und natürlich ist auch die Flanke konfigurierbar. Man muss lediglich beachten, dass sich gleiche Pins an verschiedenen Ports gegenseitig ausschließen – also nicht gleichzeitig als externe Interrupt-Eingänge genutzt werden können. Bei unserem Breakout-Board sind es letztlich die Pins PA0 und PA1, die auf externe Interrupts konfiguriert werden. Damit ließen sich dann zwei schnelle Zähler realisieren oder ein Zähler, der ein richtungsabhängiges Quadratursignal decodiert. Alternativ könnte man damit auch eine Zeitmessung zwischen zwei Lichtschranken realisieren.

2.4.6 Analoge Eingabe

Eine besondere Stärke von Mikrocontrollern sind Analogeingänge, und unser STM32F412 hat einen sehr guten *Analog-Digital-Umsetzer* (ADC) mit bis zu 16 Eingängen und 12 Bit Auflösung integriert.

Werfen wir zunächst einen genaueren Blick auf den Analog-Digital-Umsetzer und betrachten wir dessen Arbeitsweise. Im Grunde besteht der ADC tatsächlich aus einem *Digital-Analog-Umsetzer* (DAC) mit einem *Komparator*. Die Arbeitsweise nennt sich *sukzessive Approximation*. Dabei wird am DAC ein Analogwert eingestellt, und mit dem Komparator wird lediglich verglichen, ob der Analogwert am Eingang größer oder kleiner ist. Das Verfahren beginnt mit dem höchstwertigen Bit, was dem halben Messbereich entspricht, und nähert sich sukzessive immer weiter in Richtung der niederwertigen Bits an. Jedes weitere Bit halbiert den verbleibenden Messbereich, sodass die Approximation *logarithmisch konvergiert* – nach zwölf Iterationen hat der AD-Wandler also einen 12-Bit-Analogwert ermittelt. Eine Auflösung von 12 Bit entspricht einem Wertebereich von 0 bis 4095. Wir werden jedoch noch sehen, dass wir die Auflösung rein rechnerisch verdoppeln können, also auf 0 bis 8192. Das ist für einen Mikrocontroller dann schon sehr gut.

Zunächst interessiert uns nur wieder die Pin-Konfiguration. Der STM32F412 hat 16 Pins, die potenziell als Analogeingänge konfiguriert werden können. Da die meisten der Pins jedoch an Port A liegen und wir von Port A vorrangig die Schnittstellen (UART, SPI) verwenden wollen, bieten sich vier Pins von Port C an. Dabei handelt es sich um die Pins PC0 bis PC3. Diese Pins entsprechen den Kanälen 10 bis 13 des ADC im Mikrocontroller. Damit hat unser Breakout-Board standardmäßig also vier Analogeingänge. Für den Fall, dass das einmal nicht ausreichen sollte, könnte man die beiden Interrupt-Eingänge INT0 und INT1 auch noch zu Analogeingängen umkonfigurieren. Letztlich stehen also bis zu sechs Analogeingänge an unserem Breakout-Board zur Verfügung. Das Umkonfigurieren ist zwar nur als Option gedacht, zumindest liegen diese Pins aber schon passend nebeneinander.

Welchen Spannungsbereich messen die Analogeingänge überhaupt? Der ADC im STM32 hat eine eigene, getrennte Spannungsversorgung (VDDA). Das ist darin begründet, dass insbesondere digitale Schaltungen viele Störungen auf der Betriebsspannung hervorrufen. Diese Störungen möchte man von der Versorgung des ADC fernhalten.

Versorgungsspannung des ADC

VDDA ist die positive Versorgungsspannung des ADC.

VREF ist die Referenzspannung des ADC.

VREF ist intern mit VDDA verbunden!

Die Analogeingänge haben einen Messbereich von 0 V bis VDDA. Laut Datenblatt darf man VDDA beim STM32F412 jedoch nicht beliebig wählen, was sicher so zu erklären ist, dass es ja intern im Chip Verbindungen zwischen dem ADC und dem Prozessorbus gibt.

Wir wählen der Einfachheit halber für VDDA dieselben 3.3 V wie für den STM32 auch. Damit sich jedoch die digitalen Störungen nicht direkt in den ADC übertragen, schalten wir ein *LC-Tiefpassfilter* zwischen die 3.3 V und VDDA. Ein LC-Tiefpass besteht aus zwei aktiven Bauelementen, einer Spule und einem Kondensator – es ist also ein Filter *zweiter Ordnung* und besitzt daher eine größere Steilheit. Die Dämpfung beträgt 40 dB pro Dekade. Gleichspannungsmäßig hat die Spule dagegen nur einen sehr geringen Widerstand, sodass wir kaum einen Spannungsabfall hervorrufen. Aus diesen beiden Gründen sind LC-Filter in Versorgungsleitungen immer besser und RC-Kombinationen vorzuziehen.

In der Regel liefern jedoch Sensoren, wie beispielsweise Temperaturfühler, selten einen ideal passenden Spannungswert. Es wird daher immer nötig sein, eine Anpassung vorzunehmen, sodass der Sensorwert in einem definierten Bereich genau auf den Analogbereich von 0 bis 3.3 V abgebildet wird. Dazu eignen sich *Operationsverstärker* sehr gut, und speziell wären das die sogenannten *Instrumentenverstärker*. Alternativ kann man natürlich auch auf Standardsignale setzen, wie beispielsweise 0 bis 10 V oder 0 bis 20 mA, die in der Automatisierungstechnik sehr verbreitet sind. Eine Anpassung erfordert dafür lediglich ein oder zwei Widerstände als Spannungsteiler bzw. einen *Shunt*-Widerstand als *Bürde* für den Strom.

Da wir den AD-Wandler des STM32 nun mit 3.3 V versorgen, dürfen die Analogeingänge keinesfalls eine höhere Spannung erhalten – auch nicht kurzzeitig. Das könnte zur Zerstörung des Eingangs oder sogar des ganzen Chips führen. Es ist daher ratsam, unmittelbar vor jedem Analogeingang eine Schutzschaltung, bestehend aus einem Längswiderstand (10 kOhm), einem Kondensator (100 nF) und zwei Schottky-Dioden (z. B. BAT43), vorzusehen.

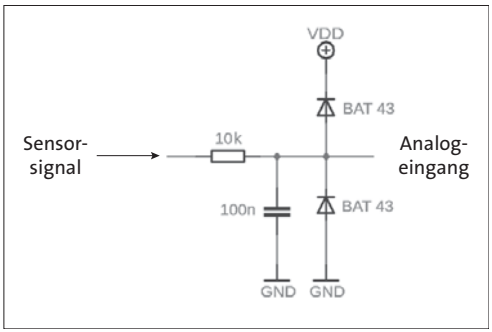

Abbildung 2.7 Schutzschaltung für Analogeingänge

Diese Schaltung erfüllt gleich zwei Aufgaben. Wird das Sensorsignal einmal größer als VDD = 3.3V, dann wird die obere Diode leitend und begrenzt den Spannungswert am

Analogeingang auf VDD + 0.3V. Das Gleiche passiert, sollte das Sensorsignal kleiner als die Masse werden. In beiden Fällen begrenzt der Längswiderstand zudem den Strom durch die Schutzdiode. Die zweite Aufgabe besteht darin, das Sensorsignal etwas zu glätten, d. h. höhere Frequenzanteile herauszufiltern. Der Widerstand und der Kondensator bilden dazu einen *RC-Tiefpass*. Die benötigte Spannung VDD ist an Pin 39 unseres Breakout-Boards verfügbar.

Bei der Programmierung der Analogwertmessung werden wir noch sehen, dass man trotz einer einfachen Schaltungstechnik sehr gute Ergebnisse erzielen kann – ein bisschen wird uns dabei auch die Mathematik helfen.

2.4.7 Analoge Ausgabe

Der STM32F412 verfügt leider nicht über einen DA-Wandler. Es ist daher nicht möglich, an einem der Pins eine Spannung auszugeben. Das ist jedoch kein nennenswerter Nachteil, weil man eine analoge Ausgabe dennoch sehr einfach realisieren kann. Das Zauberwort heißt *Pulsweitenmodulation* (PWM). Ein pulsweitenmoduliertes Signal ist eine generische digitale Signalform, die den Wert einer Mess- oder Stellgröße durch das zeitliche Verhältnis (*Duty Cycle*) zweier Signalpegel beschreibt. Der Vorteil des an sich digitalen PWM-Signals liegt in seiner einfachen Umwandlung in eine analoge Größe mittels *Integration*. Nimmt man beispielsweise einen Kondensator als integrierendes Element, erhält man eine analoge Spannung, mit einer Spule, z. B. einem Motor, kann man dagegen den Strom integrieren. Die Kombination aus einem verlustfrei übertragbaren digitalen Signal und der einfachen Umwandlung durch Integration macht das PWM-Signal so universell.

Unsere digitalen Ausgänge sind alle in der Lage, ein PWM-Signal zu erzeugen. So gesehen wären alle sechs Ausgänge unseres Breakout-Boards grundsätzlich auch als analoge Ausgänge verwendbar. Da die Anzahl der digitalen Ausgänge aber ohnehin schon begrenzt ist, wollen wir einen expliziten PWM-Ausgang zur Erzeugung analoge Signale vorsehen.

Der Pin AOUT unseres Breakout-Boards geht an PB10 des STM32F412, das mit Kanal 3 von Timer 2 verbunden werden kann. Damit besteht die Möglichkeit, an diesem Pin kontinuierlich ein PWM-Signal mit einer Frequenz f_{PWM} = 10 kHz auszugeben. Da das eine Funktionalität des Timers ist, benötigt diese PWM-Ausgabe keinerlei Rechenzeit – einmal initialisiert, wird kontinuierlich ein exaktes PWM-Signal ausgegeben.

Die nachfolgende Schaltung wandelt das PWM-Signal an AOUT in eine analoge Spannung im Bereich von 0 bis 10 V um. Dieser Spannungsbereich ist Standard und in der Automatisierungstechnik sehr stark verbreitet. Die Schaltung besteht aus zwei Teilen –

dem RC-Glied, das die Integration der Spannung übernimmt, und dem Verstärker, der den Spannungsbereich anpasst. Dabei ist hier nicht dargestellt, dass der *Operationsverstärker* (OPV) eine Versorgungsspannung von mindesten 12 V haben muss.

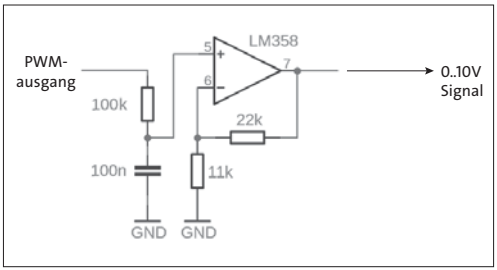

Abbildung 2.8 Einfacher Analogausgang mit 0 bis 10 V Standardsignal

Schauen wir uns die Funktion dieser Schaltung genauer an. Das RC-Glied wird hier zur Integration des PWM-Signals verwendet. Dabei gehen zwei Größen in die Analogwandlung ein. Einerseits stellt die Amplitude des PWM-Signals, U_{PWM} = 3.3 V, den Maximalwert dar, auf den sich der Kondensator aufladen kann. Andererseits wirkt sich das Tastverhältnis des PWM-Signals auf den Spannungswert am Kondensator aus. Nehmen wir ein Tastverhältnis von 50 % an – *High*- und *Low*-Zeit des PWM-Signals sollen gleich sein. Der Kondensator wird also genauso lange geladen, wie er anschließend wieder entladen wird. Aufgrund des Verlaufs von Lade- und Entladekurve wird sich dabei allerdings ein Spannungswert um die 1.65 V am Kondensator einstellen! Je nach Größe des Kondensators bzw. auch des Widerstands bleibt allerdings eine mehr oder weniger große *Ripple-Spannung*. Macht man den Kondensator nur groß genug, wird die Ripple-Spannung gegen null gehen. Das heißt, ein einfaches RC-Glied reicht prinzipiell aus, um aus einem PWM-Signal eine analoge Spannung im Bereich von 0 bis U_{PWM} zu generieren. Der nachfolgende *nicht invertierende Verstärker* ändert lediglich den Spannungsbereich auf 0 bis 10 V.

Die Ripple-Spannung kann bei einem einfachen RC-Glied insofern zum Problem werden, da man beim Vergrößern der Kapazität auch die Hochlaufzeit verlängert. Salopp gesagt, wird unser einfacher DA-Wandler entweder sehr träge und die Ausgabe veränderlicher Analogwerte damit schwieriger, oder man muss mit einer hohen Ripple-Spannung leben. Die Lösung für dieses Dilemma besteht darin, das einfache RC-Glied, ein Tiefpass 1. Ordnung, durch einen Tiefpass höherer Ordnung zu ersetzen.

An der Stelle nimmt man einen *aktiven Tiefpass* mit Operationsverstärker. Wir lassen die Schaltung aus Abbildung 2.8 quasi unverändert, schalten jedoch noch einen Tiefpass 2. Ordnung davor (siehe Quelle 7 im Literaturverzeichnis).

Kapitel 4
Betriebssystem

Eine Software, bei der man alle Fehler selbst gefunden hat, ist schließlich eine Software, die man wirklich versteht.

Immer wenn eine Software mehrere Aufgaben gleichzeitig bearbeiten muss, stellt sich die Frage nach einem Betriebssystem. Bei einem PC ist die Antwort eindeutig, man will einen Text bearbeiten und vielleicht gleichzeitig Musik aus dem Internet streamen. Wir haben es aber mit einem Mikrocontroller zu tun, Textverarbeitung ist bestimmt nicht die erste Aufgabe, die einem dabei einfällt. Was sind also die Aufgaben, die unser Mikrocontroller bearbeiten muss, und brauchen wir dafür ein Betriebssystem?

Unser Breakout-Board hat mehrere Schnittstellen, wie die beiden UARTs, die Modbus-Schnittstelle und nicht zu vergessen die analogen und digitalen Ein- und Ausgänge. Es wird gleichzeitig mehrere Datenprotokolle geben, wie etwa das Modbus-Protokoll, aber auch andere Kommunikationsprotokolle, und es wird natürlich Anwendungslogik für die I/Os geben. Wir werden also eine Software schreiben müssen, die sich gleichzeitig um alle diese Schnittstellen kümmern kann.

Wir wollen die Software in C schreiben, und ganz prinzipiell besteht ein C-Programm nur aus einer `main()`-Funktion. Wenn wir da alle Aufgaben parallel und ohne Unterstützung durch ein Betriebssystem abarbeiten wollten, sähe das in etwa so aus.

```
// --- Multithreading? ---
VOID main()
{
    for (;;)
    {
        HandleIO();
        HandleComm();
        HandleModbus();
    }
}
```

Listing 4.1 Naiver Ansatz für ein einfaches Multithreading

Diese Variante funktioniert zwar, aber nur, solange die Summe der Ausführungszeiten aller Funktionen kleiner als die maximal zulässige Reaktionszeit der häufigsten Aufgabe ist. Würde man etwa einen I/O-Pin alle 100 ms umschalten wollen, müssten sämtliche Funktionen in der Zeit einmal durchgelaufen sein. Das ist natürlich möglich und bei vielen kleineren Mikrocontroller-Applikationen sicher schon oft so realisiert worden. In der Regel wird man mit diesem Ansatz einen Mikrocontroller aber nicht optimal ausnutzen können, weil diese Lösung nicht skalierbar ist. Was würde passieren, wenn der Pin später irgendwann nur alle 10 ms umschalten soll?

Die Ausführung von mehreren Aufgaben (*Multithreading*) ist aber nur ein Aspekt, bei dem ein Betriebssystem nützlich ist. Darüber hinaus abstrahiert es die Hardware, stellt Treiber, Datenpuffer oder sogar Dateien zur Verfügung. In der Regel ist die Softwareentwicklung unter Nutzung eines Betriebssystems deutlich komfortabler als ohne. Im Bereich Embedded Systems gibt es einige etablierte Betriebssysteme, wie beispielsweise *FreeRTOS*, *MbedOS* oder *RIOT*, die alle auch für den STM32 erhältlich sind.

Ist es da wirklich noch notwendig oder sinnvoll, ein Betriebssystem selbst zu schreiben? Die Frage impliziert ein klares Nein, aber genau genommen lässt sie sich aufgrund der bloßen Verfügbarkeit solcher Systeme gar nicht beantworten. Wenn man eine Softwarekomponente einsetzt, legt man sich in gewisser Weise fest und geht damit quasi eine Abhängigkeit ein. Es stellen sich sofort weitreichende Fragen: Wie sind die Lizenzbedingungen? Wie bekommt man Updates? Wie erfolgt eine Fehlerbeseitigung? Wie gut ist der Support?

Demgegenüber steht natürlich der Aufwand, alles selbst zu entwickeln. Dieser Aufwand wird gern überschätzt, während die Probleme beim Einsatz fertiger Softwarekomponenten eher unterschätzt werden.

In den folgenden Abschnitten wollen wir ein minimales Betriebssystem selbst entwickeln und sehen, wie weit wir letztlich kommen. Am Ende stellen wir die obige Frage noch einmal.

4.1 Initialisierung

Unser Betriebssystem beginnt an der Adresse 0x08004000 und wird dort vom Bootloader gestartet. In Abbildung 3.1 sind die ersten fünf FLASH-Segmente dargestellt – die Adresse 0x08004000 ist der Beginn des zweiten, 16 kB großen FLASH-Segments direkt hinter dem Bootloader. Das ist deshalb wichtig, weil unser Betriebssystem an dieser Adresse ohne den Bootloader gar nicht starten würde!

Für den Fall, dass man das Betriebssystem aber debuggen möchte, muss man zunächst eine Version erstellen, die im ersten FLASH-Segment an der Adresse 0x08000000 beginnt. Dafür sind lediglich zwei kleinere Anpassungen nötig. Im Linker-Skript muss man die Startadresse des FLASH-Speichers ändern und in der Datei *System.c* den Offset der Vektortabelle löschen.

Betriebssystem ohne Bootloader

Um das Betriebssystem auch ohne den Bootloader startfähig zu machen, müssen Sie im Linker-Skript die Anfangsadresse des FLASH-Speichers (ORIGIN) auf 0x08000000 setzen und in der Datei *System.c* den Offset der Vektortabelle auf 0x0000 ändern.

Diese Änderungen bewirken, dass man ein startfähiges Betriebssystem erhält, das dann geeignet ist, um es etwa direkt aus der CubeIDE heraus zu debuggen oder auch um das Betriebssystem stand-alone, ohne Bootloader, zu verwenden.

4.1.1 PLL-Clock

Unser Mikrocontroller soll mit einem Systemtakt von 100 MHz operieren. Wir haben extern jedoch nur einen 8-MHz-Quarz – wie kann das überhaupt funktionieren? Schauen wir uns dazu ein Detail an, das heute bei praktisch allen besseren Mikrocontrollern verwendet wird – die sogenannte PLL (*Phased Locked Loop*).

Der eigentliche Systemtakt wird im Mikrocontroller von einem hochfrequenten, spannungsgesteuerten, aber freilaufenden Taktgenerator, einem VCO (*Voltage Controlled Oszillator*), erzeugt. Dieser freilaufende Takt wird zunächst durch einen programmierbaren Teiler geleitet, der im Ergebnis dann einen 8-MHz-Takt liefert. Der Quarz außen am Mikrocontroller dient nun als Referenz für einen *Phasendiskriminator*. Dabei handelt es sich um einen Komparator, der die Phasenlage von internem und externem Takt vergleicht und je nach Abweichung eine Steuerspannung zur Korrektur an den freilaufenden Taktgenerator liefert. Im Grunde ist das also ein Regelkreis, der den geteilten Systemtakt *phasensynchron* zum externen Quarztakt hält. Durch den Phasenvergleich erreicht man die gleiche Genauigkeit, die der Quarztakt liefert, mit dem Unterschied, dass der Systemtakt nun um den Teilerfaktor schneller läuft.

Die Verwendung einer PLL hat gleich mehrere Vorteile. Ganz wichtig ist dabei natürlich die Möglichkeit, den internen Systemtakt umprogrammieren zu können. Dazu muss man lediglich den programmierbaren Teiler konfigurieren, und schon erzeugt der VCO eine andere Frequenz – der Quarz muss dazu nicht geändert werden. Ein Mikrocontroller benötigt intern nicht nur einen Takt. In der Tat sind es viele verschiedene Takte, die

aber alle synchron zueinander laufen. Das kann man nur erreichen, indem man programmierbare Teiler zwischenschaltet und auch den Basistakt, also den Takt des VCO, konfigurierbar macht. Wir werden uns bei den Hardwarekomponenten noch mit den dort verfügbaren Takten und Teilern befassen.

Der zweite, ebenfalls sehr wichtige Aspekt ist der positive Einfluss auf die Aussendung elektromagnetischer Störstrahlung. Durch den niederfrequenten Quarz erzeugt man weniger Störfrequenzen, wodurch sich das Störspektrum insgesamt verbessert. Nicht zuletzt kann man auch günstigere Grundtonquarze einsetzen, die ein stabiles Schwingverhalten aufweisen.

Moment, wie kann denn der Mikrocontroller seine PLL konfigurieren, wenn die PLL doch erst benötigt wird, damit ein Programm überhaupt loslaufen kann? Dafür gibt es eine einfache und einleuchtende Erklärung – ein STM32 startet zuerst mit einem separaten internen Takt von 16 MHz – einer Art Hilfstakt. Für kostensensitive Applikationen könnte man den externen Quarz sogar weglassen und nur mit dem Hilfstakt operieren. Wenn der externe Quarz eingeschwungen und die PLL konfiguriert ist, wird durch Setzen eines Bits auf den Systemtakt umgeschaltet. Das Programm läuft dann quasi schlagartig schneller.

Schauen wir uns an, wie die PLL konfiguriert wird. Dazu gibt es in der Datei *system_stm32f4xx.c*, die zur CMSIS-Library gehört, drei Definitionen:

```
// PLL configuration
#define PLL_N           200
#define PLL_M             8
#define PLL_P             2
```

Listing 4.2 Koeffizienten für die PLL-Konfiguration

Die Konfiguration der PLL erfolgt danach anhand der folgenden Gleichung:

[4.1] $f_{SYSCLK} = HSE_VALUE : PLL_M \cdot PLL_N : PLL_P$

Der Wert HSE_VALUE wird in der Datei *stm32f4xx.h* definiert und ist die Frequenz des externen Quarzes $f_{CRYSTAL}$, in unserem Fall also 8 MHz. Setzt man nun die Werte aus Gleichung 4.1 ein, ergibt sich für die Frequenz des Systemtakts f_{SYSCLK} ein Wert von 100 MHz.

Die Funktionen `SystemInit()` und `SetSysClock()`, die ebenfalls von der CMSIS-Library bereitstellt werden, führen letztlich die Konfiguration der PLL aus. Wir rufen dazu die Funktion `SystemInit()` beim Start des Betriebssystems auf. Das passiert ziemlich direkt aus der `main()`-Funktion heraus. Man kann mit der CubeIDE diese Funktionen auch direkt debuggen und die einzelnen Schritte verfolgen. So werden in der Funktion `SystemInit()` zunächst alle Einstellungen rückgängig gemacht – die PLL-Konfiguration wird

quasi wieder in den Reset-Zustand versetzt. Das ist konsequent, weil unser Betriebssystem ja nicht das erste Programm ist, das auf dem Mikrocontroller läuft. Der Bootloader hatte seinerseits schon eine Systeminitialisierung vorgenommen, und in der Tat operiert unser Bootloader mit einem geringeren Systemtakt! Der Grund ist: Unser Bootloader kann so auch auf einem STM32F401 verwendet werden, der zwar dasselbe Speicherlayout aufweist, aber nur mit maximal 84 MHz operieren kann.

Der zweite wichtige Aufruf während der Initialisierungsphase ist der Aufruf der Funktion `SystemCoreClockUpdate()`. In dieser Funktion wird eine globale Variable mit dem Namen `SystemCoreClock` zugewiesen. Diese Variable kann zu jeder Zeit verwendet werden, um die eingestellte Frequenz des Systemtakts zu ermitteln. Für den Fall, dass man die PLL einmal anders konfigurieren möchte, sollte man anschließend mit dem Debugger prüfen, ob diese Variable die gewünschte Frequenz anzeigt.

Damit sind die PLL und der Systemtakt konfiguriert. Man kann an der Stelle gleich noch den Takt für jedes verwendete Hardwaremodul (*Peripheral Modules*) aktivieren. Nach einem Reset sind alle Module, wie beispielsweise die Ports oder die UARTs, standardmäßig deaktiviert – sie bekommen einfach noch keinen Takt. Solange ein Hardwaremodul keinen Takt bekommt, ist dessen Stromaufnahme praktisch null. In unserem Betriebssystem erfolgt die Aktivierung aller Module gleich im Anschluss an die Initialisierung des Systemtakts in der Funktion `InitSysConfig()`. Alternativ kann die Freigabe des Takts aber auch im Rahmen der Initialisierung eines jeden Moduls erfolgen.

4.1.2 GPIO-Setup

Wie Peripheriemodule mithilfe der CMSIS-Library initialisiert werden, kann man sehr gut an den GPIO-Modulen, also den I/O-Pins unseres Breakout-Boards, sehen. Das Prinzip ist aber immer ähnlich, ganz gleich, ob es sich nur um die GPIO-Pins oder um den AD-Wandler oder eine UART handelt.

Die CMSIS-Library stellt passende Strukturen zur Verfügung, die man zunächst mit Initialwerten füllt. Anschließend ruft man eine Library-Funktion auf und übergibt ihr die zuvor initialisierte Struktur. Zu jeder Strukturdeklaration gehören immer auch »sprechende« Konstanten, sodass die Initialisierung sehr gut lesbar und leicht verständlich wird. Der nachfolgende Auszug gibt einen Einblick in die GPIO-Initialisierung, wie sie in unserem Betriebssystem implementiert ist.

```
// --- I/O definitions ---

#define IN_PINS0     GPIOC
#define IN_0         GPIO_Pin_9
#define IN_1         GPIO_Pin_8
```

```
#define IN_2         GPIO_Pin_7
#define IN_3         GPIO_Pin_6
#define IN_4         GPIO_Pin_11
#define IN_5         GPIO_Pin_10
#define IN_6         GPIO_Pin_12

#define IN_PINS1     GPIOA
#define IN_7         GPIO_Pin_15
#define IN_8         GPIO_Pin_0
#define IN_9         GPIO_Pin_1

#define OUT_PINS     GPIOB
#define OUT_0        GPIO_Pin_9
#define OUT_1        GPIO_Pin_8
#define OUT_2        GPIO_Pin_7
#define OUT_3        GPIO_Pin_6
#define OUT_4        GPIO_Pin_5
#define OUT_5        GPIO_Pin_4

// --- Init GPIOs ---
VOID InitGPIO()
{
    GPIO_InitTypeDef gpio_init = {0};

    // config input pins
    gpio_init.GPIO_Pin  = IN_0 | IN_1 | IN_2 | IN_3 | ...;
    gpio_init.GPIO_Mode = GPIO_Mode_IN;
    gpio_init.GPIO_PuPd = GPIO_PuPd_DOWN;
    GPIO_Init(IN_PINS0, &gpio_init);

    gpio_init.GPIO_Pin  = IN_7 | IN_8 | IN_9;
    gpio_init.GPIO_Mode = GPIO_Mode_IN;
    gpio_init.GPIO_PuPd = GPIO_PuPd_DOWN;
    GPIO_Init(IN_PINS1, &gpio_init);

    // config output pins
    gpio_init.GPIO_Pin   = OUT_0 | OUT_1 | OUT_2 | OUT_3 | ...;
    gpio_init.GPIO_Mode  = GPIO_Mode_OUT;
    gpio_init.GPIO_OType = GPIO_OType_PP;
    GPIO_Init(OUT_PINS, &gpio_init);
}
```

Listing 4.3 Auszug aus der GPIO-Initialisierung

Die Funktion `InitGPIO()` definiert eine Variable vom Typ `GPIO_InitTypeDef`. Die Typdeklaration befindet sich in der CMSIS-Library. Dabei handelt es sich um eine Struktur, die alle Informationen aufnimmt, die für die Initialisierung eines oder auch mehrerer Pins eines Ports benötigt werden. Die Ports sind als Konstanten `GPIOA`, `GPIOB` etc. vordefiniert, ebenso wie die Pins `GPIO_Pin_0` bis `GPIO_Pin_15`. Da die Pins als Bitmasken deklariert sind, kann man sie ODER-verknüpfen und so mit einem Aufruf von `GPIO_Init()` gleich mehrere Pins identisch konfigurieren.

Da die Sprache C leider keine *Namespaces* unterstützt, sind allen Bezeichnern sogenannte *Namespace Prefixes* vorangestellt. Sämtliche Konstanten und Typen, die zur GPIO-Initialisierung gehören, beginnen stets mit `GPIO_`. Das macht die Bezeichner zwar etwas länger, hilft aber auch bei der Eingabe, weil man nur diesen Anfang tippt und sich dann mit `Strg`+Leertaste eine Liste aller möglichen Bezeichner anzeigen lässt. Achtung, die Bezeichner sind sich teils wirklich sehr ähnlich, und man verwendet schnell eine unpassende Konstante oder eine unpassende Funktion. Anschließend muss man dann herausfinden, warum das Programm nicht funktioniert.

Schauen wir uns die verwendeten Konstanten der Initialisierungsstruktur noch etwas genauer an. Jeder Pin kann entweder als Eingang oder als Ausgang konfiguriert werden. Diese Unterscheidung erfolgt über das Strukturelement `GPIO_Mode`. Bei Eingängen ist es zusätzlich ratsam, den *Pull-up*- oder den *Pull-down*-Widerstand zu aktivieren, damit offene Eingänge stets auf einen definierten Pegel gezogen werden. In unserem Fall ziehen wir alle Eingänge auf *Low*, aktivieren also den Pull-down-Widerstand. Bei den Ausgängen gibt es ebenfalls eine Spezialisierung, die die konkrete Schaltungsart des Ausgangs betrifft. Wenn ein Ausgang aktiv nach Masse und nach VDD schalten kann, ist es ein *Push-Pull*-Ausgang. Alternativ gibt es Ausgänge, die nur nach Masse schalten können; dann handelt es sich um sogenannte *Open-Drain*-Ausgänge.

Die Konfiguration der Ein- und Ausgänge ist damit auch schon abgeschlossen. Die Nutzung der CMSIS-Library macht uns diese Aufgabe wirklich einfach. Wir werden aber noch sehen, dass die Library-Funktionen nicht immer ideal sind. Speziell bei den GPIO-Zugriffen werden wir stattdessen dann auf schlankere Funktionen setzen.

4.1.3 AD-Wandler

Auch die Initialisierung der Analogeingänge beginnt zunächst wieder mit der Konfiguration der entsprechenden Port-Pins. Das sind in dem Fall die Pins PC0 bis PC3. Diese Pins werden, wie auch schon bei den digitalen Eingängen, über einen Aufruf der CMSIS-Funktion `GPIO_Init()` konfiguriert. Der wesentliche Unterschied besteht in dem Wert `GPIO_Mode_AN`, der initial dem Strukturelement `GPIO_Mode` zugewiesen wird. Mit nur

einem Aufruf werden so alle vier Pins zu analogen Eingängen. Analoge Eingänge sollten weder einen Pull-up- noch einen Pull-down-Widerstand erhalten. Das würde möglicherweise zu einer Verfälschung der Messwerte führen.

```
// --- ADC definitions ---

#define ADC_PORT   GPIOC
#define ADC_0      GPIO_Pin_0
#define ADC_1      GPIO_Pin_1
#define ADC_2      GPIO_Pin_2
#define ADC_3      GPIO_Pin_3

// --- Init ADC ---
VOID InitADC()
{
    GPIO_InitTypeDef gpio_init = {0};
    ADC_InitTypeDef adc_init = {0};
    ADC_CommonInitTypeDef adc_common = {0};

    // configure analog pins
    gpio_init.GPIO_Pin  = ADC_0 | ADC_1 | ADC_2 | ADC_3;
    gpio_init.GPIO_Mode = GPIO_Mode_AN;
    gpio_init.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(ADC_PORT, &gpio_init);

    // setup common adc params
    adc_common.ADC_Mode = ADC_Mode_Independent;
    adc_common.ADC_Prescaler = ADC_Prescaler_Div2;
    adc_common.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_5Cycles;
    ADC_CommonInit(&adc_common);

    // setup adc1 parameters
    adc_init.ADC_Resolution = ADC_Resolution_12b;
    adc_init.ADC_ContinuousConvMode = DISABLE;
    adc_init.ADC_ScanConvMode = DISABLE;
    adc_init.ADC_NbrOfConversion = 1;
    ADC_Init(ADC1, &adc_init);

    // enable adc
    ADC_Cmd(ADC1, ENABLE);
}
```

Listing 4.4 Initialisierung des AD-Wandlers

Der zweite Schritt ist die Initialisierung der gemeinsamen Einstellungen, die für alle Analog-Digital-Wandler gelten. Unser STM32F412 verfügt tatsächlich nur über einen AD-Wandler, und im Grunde können wir bei einem STM32F412 auch nur den Vorteiler (`ADC_Prescaler`) für den AD-Wandler konfigurieren. Hier stellen wir den kleinstmöglichen Teiler ein, was dem Default-Wert entspricht. Weitere Konfigurationsmöglichkeiten gibt es bei diesem Mikrocontroller nicht. Der Schritt wäre so gesehen nicht unbedingt nötig. An der Stelle muss man aber sagen, dass die CMSIS-Library für eine Vielzahl von STM32-Varianten erstellt worden ist. Im Sinne einer leichteren Portierbarkeit ergibt es also schon Sinn, die eigene Software auch möglichst generisch zu halten. Würden wir unser Betriebssystem später einmal portieren wollen, etwa auf einen STM32F446, der über drei AD-Wandler verfügt, müssten wir zumindest an der Stelle nichts ändern.

Im letzten Schritt wird noch der AD-Wandler an sich, hier ADC1, konfiguriert. Die wesentliche Einstellung betrifft die Auflösung von 12 Bit. Man kann die Auflösung zwar auch reduzieren, wenn es auf eine möglichst kurze Wandlungszeit ankommt, in der Regel wird man aber die höchste Auflösung wählen. Zudem bieten die AD-Wandler aller STM32-Mikrocontroller auch die Möglichkeit, ausgewählte Eingänge kontinuierlich zu wandeln. Das ergibt in unserem Fall jedoch keinen Sinn, weil die Werte nur nach Bedarf gelesen werden. Schließlich muss der AD-Wandler mit der Funktion `ADC_Cmd()` nur noch eingeschaltet werden, und fortan kann man jederzeit eine Messung starten.

Für die eigentliche Messung werden wir noch eine Funktion erstellen, die dann den AD-Wandler verwendet.

4.1.4 DA-Wandler

Bisher haben wir uns mit der Konfiguration der Ein- und Ausgänge befasst sowie mit der Konfiguration von Analogeingängen. Für die Konfiguration des PWM-Ausgangs für den DA-Wandler müssen wir uns mit der alternativen Konfiguration von Pins beschäftigen. Theoretisch kann jeder Pin 15 alternative Funktionen (AF, *Alternate Functions*) zugewiesen bekommen. Die tatsächlich zu einem Pin verfügbaren Funktionen findet man in Tabellen im Datenblatt des jeweiligen Controllertyps. Eine solche Tabelle hatten wir bereits bei der Auswahl der I/O-Pins verwendet. In Abbildung 2.6 sind beispielsweise alle AFs an Port B aufgeführt.

Mit der Funktion `GPIO_PinAFConfig()` weisen wir einem Pin eine verfügbare Funktion zu. Man könnte umgekehrt auch sagen, dass der Pin einer konkreten Hardware zugewiesen wird. Letztlich richtet sich die Funktion des Pins dann nach der Hardware. Dem Pin selbst muss man bei der Konfiguration den Modus `GPIO_Mode_AF` mitgeben.

```
// --- Defines ---
// PB10 = PWM

#define DAC_PORT            GPIOB
#define DAC_PIN             GPIO_Pin_10
#define DAC_PIN_SRC         GPIO_PinSource10

// --- Init DAC ---
VOID InitDAC()
{
    GPIO_InitTypeDef gpio_init = {0};
    TIM_OCInitTypeDef tmr_ocinit = {0};
    TIM_TimeBaseInitTypeDef tmr_setup = {0};

    // alternate function PWM
    GPIO_PinAFConfig(DAC_PORT, DAC_PIN_SRC, GPIO_AF_TIM2);

    // setup pin as AF output
    gpio_init.GPIO_Pin = DAC_PIN;
    gpio_init.GPIO_Mode = GPIO_Mode_AF;
    gpio_init.GPIO_OType = GPIO_OType_PP;
    GPIO_Init(DAC_PORT, &gpio_init);

    // setup timer base
    tmr_setup.TIM_Period = DAC_RANGE;
    tmr_setup.TIM_ClockDivision = TIM_CKD_DIV1;
    tmr_setup.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM2, &tmr_setup);

    // setup output compare
    tmr_ocinit.TIM_OCMode = TIM_OCMode_PWM1;
    tmr_ocinit.TIM_OCPolarity = TIM_OCPolarity_High;
    tmr_ocinit.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OC3Init(TIM2, &tmr_ocinit);

    // channel 3 auto reload
    TIM_OC3PreloadConfig(TIM2, TIM_OCPreload_Enable);

    // timer 2 auto reload
    TIM_ARRPreloadConfig(TIM2, ENABLE);
```

```
    // enable timer
    TIM_Cmd(TIM2, ENABLE);
}
```

Listing 4.5 Initialisierung des PWM-Ausgangs

Mit der obigen Initialisierung wird Pin PB10 dem Timer 2 zugewiesen. Der Pin ist dann automatisch auf den Kanal 3 des Timers verbunden. Schauen wir uns genauer an, wie die Timer funktionieren.

Unser STM32F412 verfügt insgesamt über acht 16-Bit- und vier 32-Bit-Timer, alle jeweils mit vier separaten Kanälen. Es handelt es sich dem Prinzip nach bei allen Timern um *Modulo-Zähler*. Das bedeutet, man kann einen Wertebereich vorgeben, und der Timer durchläuft, abhängig von seiner Konfiguration, diesen Wertebereich vorwärts oder rückwärts. Die Parameter, die wir daher Timer 2 mit der Funktion `TIM_TimeBaseInit()` zuweisen, umfassen also zunächst die Periode, den Wertebereich des Zählers, sowie die Zählrichtung. Die Frage lautet natürlich, wie schnell Timer 2 nun eigentlich zählt. Timer 2 ist mit dem *Advanced Peripheral Bus 1* (APB1) verbunden, der mit einer Frequenz von 50 MHz getaktet wird. Die Timer an den beiden Peripheriebussen APB1 und APB2 werden immer mit der gleichen Taktfrequenz betrieben wie der Bus auch. Man kann bei Timer 2 zusätzlich noch einen Vorteiler auswählen, um die Taktfrequenz zu reduzieren. Wir benötigen den Vorteiler jedoch nicht, denn bei einer Periode von 5000 läuft der Timer in genau 100 µs einmal durch. Das entspricht der gewünschten Frequenz von 10 kHz, für die der DA-Tiefpass ausgelegt ist. Damit ist die Initialisierung der Zeitbasis bereits abgeschlossen.

Die Konfiguration des PWM-Kanals ist ähnlich einfach. Da die Erzeugung von PWM-Signalen eine Standardaufgabe ist, gibt es speziell dafür vordefinierte Konstanten. So lässt sich mit dem `OCMode` (*Output Compare Mode*) festlegen, ob das PWM-Signal *active high* oder *active low* sein soll. Mit der Konstanten `TIM_OCMode_PWM1` legen wir *active high* fest. Bei einem Overflow des Zählers wird der Kanal nun immer auf High-Pegel geschaltet und beim nächsten *Output Compare Event* wieder auf Low-Pegel gezogen. Das passiert alles in der Hardware des Timers und belegt keinerlei Rechenzeit.

Es bleiben noch zwei Zeilen der Initialisierung des Timers, die eine wichtige Funktion haben. Für unseren DA-Wandler wären beide Zeilen nicht zwingend erforderlich, aber deren Verständnis ist wichtig. Um das PWM-Tastverhältnis zu ändern, müssen wir lediglich das *Output Compare Register* auf einen neuen Wert setzen, und schon wird eine andere Pulsbreite generiert. So weit stimmt es. Doch was passiert, wenn wir das Compare Register unmittelbar vor einem Compare Event zurücksetzen würden, vor den aktuellen Zählerwert? Wir würden quasi ein Compare Event übergehen – es würde also nicht statt-

finden! Der Ausgang bliebe bis zum nächsten Compare auf high. Damit das nicht passiert, kann man, anstatt das Compare Register direkt zu beschreiben, in ein *Preload Register* schreiben. Der Timer übernimmt das Preload Register dann immer erst bei einem Overflow. Somit ist sichergestellt, dass das Compare Event nicht übergangen werden kann. Mit der Funktion `TIM_OC3PreloadConfig()` aktiviert man das Preload Register sowie die synchronisierte Übernahme daraus. Es gibt aber auch Fälle, in denen man dieses Verhalten nicht möchte. Das ist beispielsweise dann der Fall, wenn man das Compare Register per Interrupt weiterrückt, um mehrere Compare Events auszulösen. In dem Fall muss man natürlich direkt in das Compare Register schreiben können. Das Preload Register ist dabei kein weiteres Register, es wird einfach als das Compare Register adressiert, nur dass die Hardware des Timers intern eine Art »Schattenregister« dafür vorhält. Die Timer des STM32 sind wahre Wunderwerke, und ein PWM-Signal zu generieren, ist so, als würde man sich von einem Musiker *Alle meine Entchen* wünschen.

4.1.5 SPI-Systembus

Mit der Konfiguration von *Alternate Function Pins* haben wir uns bereits beim PWM-Ausgang für die Analogwertausgabe befasst. Das Prinzip benötigen wir jetzt auch wieder. Die Pins der SPI-Schnittstelle sind in dem Fall auch wieder AF-Pins – deren Konfiguration gleich für alle drei Signale auf einmal erfolgen kann. Damit ist die Konfiguration der SPI-Pins auch schon abgeschlossen. Die SPI selbst ist nicht ganz so einfach, aber schauen wir uns die möglichen Parameter im Einzelnen an.

```
// --- Defines ---
// PA5  = SCL
// PA6  = SSI (MISO)
// PA7  = SSO (MOSI)

#define SPI_PINS            GPIOA
#define SPI_SCL             GPIO_Pin_5
#define SPI_SSI             GPIO_Pin_6
#define SPI_SSO             GPIO_Pin_7

#define SPI_SCL_SRC         GPIO_PinSource5
#define SPI_SSI_SRC         GPIO_PinSource6
#define SPI_SSO_SRC         GPIO_PinSource7

// --- Init SPI1 ---
VOID InitSPI()
{
```

```
    SPI_InitTypeDef spi_init = {0};
    GPIO_InitTypeDef gpio_init = {0};

    // config alternate function SPI1
    GPIO_PinAFConfig(SPI_PINS, SPI_SCL_SRC, GPIO_AF_SPI1);
    GPIO_PinAFConfig(SPI_PINS, SPI_SSI_SRC, GPIO_AF_SPI1);
    GPIO_PinAFConfig(SPI_PINS, SPI_SSO_SRC, GPIO_AF_SPI1);

    // config SPI1 pins
    gpio_init.GPIO_Pin     = SPI_SCL | SPI_SSI | SPI_SSO;
    gpio_init.GPIO_Mode    = GPIO_Mode_AF;
    gpio_init.GPIO_OType   = GPIO_OType_PP;
    GPIO_Init(SPI_PINS, &gpio_init);

    // spi data setup
    spi_init.SPI_Mode      = SPI_Mode_Master;
    spi_init.SPI_DataSize  = SPI_DataSize_8b;
    spi_init.SPI_FirstBit  = SPI_FirstBit_MSB;
    spi_init.SPI_CPOL      = SPI_CPOL_High;
    spi_init.SPI_CPHA      = SPI_CPHA_2Edge;
    spi_init.SPI_NSS       = SPI_NSS_Soft | SPI_NSSInternalSoft_Set;
    spi_init.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    spi_init.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;
    SPI_Init(SPI1, &spi_init);

    // enable SPI1
    SPI_Cmd(SPI1, ENABLE);
}
```

Listing 4.6 Initialisierung des SPI-Systembusses

Das SPI-Modul auf dem Mikrocontroller ist sehr flexibel konfigurierbar. Das liegt nicht zuletzt auch daran, dass die SPI nicht ganz so strikt spezifiziert wurde wie etwa eine I^2C-Schnittstelle. Es gibt also mehrere Freiheitsgrade, die man einstellen muss, damit eine SPI funktioniert.

Bei einem SPI-Bus darf nur der Master eine Übertragung starten und den Bustakt generieren. Unser STM32 fungiert als Busmaster. Diese Einstellung wird über das Element `SPI_Mode` zugewiesen – daher ist der erste Parameter `SPI_Mode_Master`. Auch der zweite Parameter ist einleuchtend. Wir wollen Daten immer im Byte-Format austauschen und setzen daher `SPI_DataSize` auf `SPI_DataSize_8b`. Schließlich muss man sogar die *Bit-*

Order konfigurieren – hier setzen wir die Reihenfolge auf *Most Significant Bit first* (MSB first) mit dem Parameter `SPI_FirstBit_MSB`. Bis hierhin handelt es sich quasi um die Grundeinstellungen.

Darüber hinaus gibt es zwei weitere Parameter, die durchaus für Verwirrung sorgen können und bestimmt schon oft Grund für eine fieberhafte Fehlersuche waren. Es geht um die beiden Parameter *Clock Polarity* (`CPOL`) und *Clock Phase* (`CPHA`). Was sagen diese beiden Parameter aus?

Wenn der Busmaster, also unser Mikrocontroller, eine Übertragung beginnt, setzt er zunächst die *Select-Leitung* des entsprechenden Teilnehmers auf *low*. Der so adressierte Teilnehmer schaltet sich unmittelbar an den Bus und folgt nun dem Taktsignal des Masters. Welchen Pegel muss der Master aber gerade zu Beginn, wenn er den Teilnehmer adressiert, auf der Taktleitung ausgeben? Die Antwort auf diese Frage ist die *Clock Polarity* oder `CPOL`. Die zweite Frage betrifft die Phasenlage des Taktsignals. Bei welcher Flanke des Taktsignals werden die Daten in Master und Slave übernommen? Die Antwort auf diese Frage ist die *Clock Phase* oder `CPHA`. Die jeweils zwei möglichen Antworten führen insgesamt zu vier möglichen Modi.

Mode	CPOL	CPHA	Bedeutung
0	LOW	1EDGE	SCL low, Übernahme bei steigender Flanke
1	LOW	2EDGE	SCL low, Übernahme bei fallender Flanke
2	HIGH	1EDGE	SCL high, Übernahme bei fallender Flanke
3	HIGH	2EDGE	SCL high, Übernahme bei steigender Flanke

Tabelle 4.1 SPI-Modi und ihre Bedeutung

Beim häufigsten Modus, Mode 0, würde der Master im Moment der Adressierung einen Low-Pegel auf der Taktleitung ausgeben. Die erste Flanke wäre dann logischerweise eine steigende Flanke, bei der Master und Slave jeweils die anliegenden Datenbits übernehmen. Mit der zweiten, also mit der fallenden Flanke würden dann die nächsten Bits ausgegeben usw.

Wenn im Datenblatt eines Busteilnehmers der SPI-Modus als Nummer steht, findet man in Tabelle 4.1 die beiden Parameter für die SPI-Konfiguration. Meist findet man in Datenblättern jedoch eher Takttelegramme. Dann schaut man darin zuerst auf den Ruhepegel des Takts und dann auf die Taktflanke, bei der die Datenbits stabil sind. So kann man die beiden Parameter für die SPI-Konfiguration leicht ermitteln. Es gibt natürlich noch die *Brute-Force-Methode* – einfach alle vier Modi durchprobieren.

Die Hersteller von Schaltkreisen mit SPI-Schnittstelle versuchen, ihre Chips so tolerant wie möglich zu gestalten. Es kann also gut sein, dass sogar zwei Modi gleichermaßen funktionieren – etwa 0 und 3 oder 1 und 2, weil diese Modi jeweils die Daten bei der gleichen Taktflanke übernehmen. Wir verwenden hier Modus 3, weil er sich erfahrungsgemäß bei den meisten SPI-Schaltkreisen als funktional erwiesen hat. Modus 0 und Modus 3 unterscheiden sich lediglich durch einen redundanten Flankenwechsel unmittelbar nach dem Chip-Select, was dem Teilnehmer zusätzlich etwas Zeit verschafft, das erste Datenbit bereitzustellen.

Mit welcher Taktrate operiert unsere SPI nun überhaupt? Der Nachteil, dass eine SPI nicht so strikt definiert ist, ist gleichzeitig auch ein Vorteil. Bei einer SPI-Schnittstelle ist die Taktrate nämlich frei wählbar und wird nur durch die maximal möglichen Taktraten der Teilnehmer begrenzt. Deshalb können wir die höchste gemeinsame Taktrate einstellen oder jeden Teilnehmer mit dessen maximaler Taktrate ansprechen. Der EEPROM erlaubt dabei eine Taktrate von bis zu 25 MHz, das LCD-Display aber nur maximal 20 MHz. Wir wählen also eine Taktrate, die entweder kleiner oder gleich der kleinsten mögliche Frequenz ist.

Die Einstellung der Taktrate erfolgt dabei in Form eines Teilerfaktors bezogen auf den Bustakt, an dem das Modul intern angeschlossen ist. Das Modul SPI1 ist beim STM32F4 am APB2-Bus (*Advanced Peripheral Bus*) angeschlossen. Dieser Bus ist ein *High-Speed-Bus* und wird äquivalent mit der System-Clock von 100 MHz getaktet. Allerdings erhält die SPI noch mal einen davon abgeleiteten peripheren Takt f_{PCLK2} von:

$$f_{PCLK2} = f_{SYSCLK} : 2 \qquad [4.2]$$

... also mit einer Frequenz von 50 MHz. Stellen wir dazu noch den Teilerfaktor von 4 für den SPI-Systembus ein, erhalten wir im Ergebnis eine Taktrate der SPI von $f_{PCLK2} : 4$ gleich 12.5 MHz. Für den EEPROM allein könnten wir auch den Teilerfaktor 2 wählen. Mit der resultierenden Taktrate von 25 MHz dürften wir aber das LCD-Display nicht betreiben. Für eine jeweils optimale Performance müssten wir die SPI stets umkonfigurieren. Diesen Aufwand wollen wir allerdings gar nicht betreiben – 12.5 MHz ist also die gewählte Taktrate.

4.1.6 NVI-Controller

Jeder STM32 besitzt einen sogenannten *Nested Vectored Interrupt Controller* (NVIC), der das Verhalten bei einem aufgetretenen Interrupt steuert. Genauer gesagt, ist der NVIC ein Merkmal aller Cortex-M-Mikrocontroller von ARM. Vor der Einführung der Cortex-M-Architektur im Jahr 2004 war die Interrupt-Verarbeitung der ARM-Mikrocontroller weniger effizient. Es gab keine direkte Vektortabelle, nur eine Liste mit Sprungbefehlen,

und auch die Anbindung an den CPU-Kern war nicht so effektiv. Da aber insbesondere Mikrocontroller ihre Vorteile aus einer effizienten Behandlung von Interrupt-Ereignissen ziehen, stellen die *ARM-Cortex-M-Mikrocontroller* einen echten Fortschritt dar.

Um die Möglichkeiten des NVIC zu verdeutlichen, soll Abbildung 4.1 eine bessere Vorstellung ermöglichen. Es gibt grundsätzlich die Möglichkeit, jeder Interrupt-Quelle eine Priorität zuzuordnen. Das bedeutet, dass jeder Interrupt natürlich die `main()`-Funktion unterbrechen kann, aber es kann auch ein Interrupt mit höherer Priorität einen bereits laufenden Interrupt niederer Priorität unterbrechen.

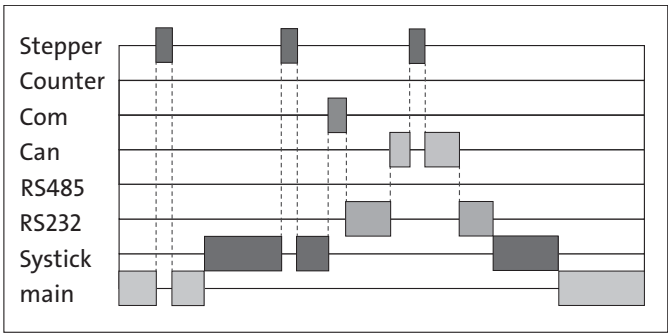

Abbildung 4.1 Nested Interrupts bei unterschiedlichen Prioritäten

Die niedrigste *Interrupt-Priorität* hat der System-Timer (*Systick*). Dieser Timer spielt eine wichtige Rolle, wenn es darum geht, ein Multitasking-System zu entwickeln. Nehmen wir für den Moment aber nur an, dass der *Systick-Interrupt* einen Millisekundenzähler inkrementiert. Dazu muss der Systick zwar die `main()`-Funktion unterbrechen, aber andere Interrupts nicht. Das andere Extrem wäre beispielsweise die Steuerung eines Schrittmotors. Die einzelnen Schritte des Motors müssen zeitlich sehr exakt ausgegeben werden – anderenfalls macht ein Schrittmotor entweder Geräusche, oder er verliert sogar Schritte. Daher bekommt der Stepper-Interrupt die höchste Priorität und wird dadurch jeden anderen laufenden Interrupt unterbrechen. Zuletzt noch ein Beispiel für eine mittlere Priorität, wie etwa eine serielle Schnittstelle. Bei einer Baudrate von 115.200 Baud mit dem Format 8N1 würde alle 90 µs ein neues Zeichen empfangen werden. Damit kein Zeichen verloren geht, muss der Interrupt innerhalb dieser Zeit ausgeführt werden. Wäre in der Zeit aber der Schrittmotor wieder dran, würde dieser den Empfangsinterrupt unterbrechen. Man kann sich noch weitere Interrupt-Quellen vorstellen und deren Priorität relativ zu den anderen Quellen einsortieren. Im Ergebnis erhält man dann ein *prioritätsgesteuertes präemptives Interrupt-Verhalten*. Das ist genau das, was der NVIC leisten kann.

Einer der größten Vorteile der Cortex-M-Architektur ist dabei sicher der, dass Interrupt-Routinen exakt so von der Hardware aufgerufen werden, wie es auch der C-Compiler machen würde. Das ist nicht bei allen Mikrocontrollern der Fall, erleichtert die Programmierung jedoch ungemein. Das bedeutet aber auch, dass bei jedem Interrupt zuerst die Register auf den Stack gesichert werden, für die der Aufrufer (*Caller*) – in dem Fall also der Interrupt-Controller – zuständig ist. Ohne Floating-Point-Hardware sind das acht Register (R0 bis R3, R12, R14, R15, PSR), also 32 Byte, die ein Interrupt an Stackspeicher benötigt. Wenn die *Floating-Point-Unit* (FPU) aktiviert ist, sind es sogar 26 Register, also schon 104 Byte!

An der Stelle ergibt sich zu Recht die Frage, wie viele *Nested Interrupts* es denn nun maximal gibt. Davon hängt ja direkt ab, wie viel Stackspeicher im ungünstigsten Fall benötigt werden würde. Natürlich ist das bei allen STM32-Mikrocontrollern konfigurierbar. Dazu gibt es vier Bits, die man zwei *Prioritätsklassen* zuordnen kann, entweder der *Preemption-Priority* oder der *Sub-Priority*. Eine höhere Preemption-Priority unterbricht dabei eine niedrigere Preemption-Priority. Die Sub-Priority regelt indes, welcher Interrupt zuerst behandelt wird, wenn es mehrere Interrupt-Anforderungen mit gleicher Preemption-Priority gibt. Das klingt vielleicht etwas kompliziert, aber mit ein paar Beispielen wird der Zusammenhang schnell klar.

Werden etwa alle vier Bits der Preemption-Priority zugeordnet, gibt es 16 Prioritätslevel und demnach maximal 16 laufende Interrupts mit zusammen 1.664 Byte möglichem Stackbedarf. In dem Fall gibt es dann keine Sub-Priorities. Ordnet man zwei Bits der Preemption-Priority zu, gibt es nur noch vier Prioritätslevel mit maximal 416 Byte Stackbedarf, und die verbleibenden zwei Bits bilden dann vier Sub-Priorities. Treten zwei Interrupts auf, die zwar die gleiche Preemption-Priority haben, aber unterschiedliche Sub-Priority, dann wird die Routine mit der höheren Sub-Priority zuerst ausgeführt. Grundsätzlich gehen aber keine Interrupts verloren.

Das lässt sich dann jeweils noch in drei Preemption-Bits und ein Sub-Bit oder ein Preemption-Bit und drei Sub-Bits aufteilen. Ordnet man alle vier Bits der Sub-Priority zu, gibt es keine Nested Interrupts mehr. Dafür hat man 16 Sub-Priorities, die dann regeln, in welcher Reihenfolge die Interrupts nacheinander abgearbeitet werden. Das ist insgesamt also sehr flexibel und erlaubt eine feine Abstimmung zwischen Stackbedarf und Latenzzeit (*Latency Time*) der verschiedenen Interrupts. Unter der Latenzzeit versteht man die Zeit, die zwischen dem auslösenden Ereignis und dem Aufruf einer zugehörigen Interrupt-Funktion vergeht. Die Latenzzeit ist jedoch keine Konstante, sondern unterliegt einer gewissen Schwankungsbreite. Diese Schwankungen werden auch als *Jitter* bezeichnet.

In einem *Echtzeitbetriebssystem* entscheidet primär die Priorität zugunsten einer geringen Latenzzeit. Daher werden wir die Interrupt-Behandlung so effektiv wie nur möglich gestalten und alle vier Bits der Preemption-Priority zuordnen. Jede Interrupt-Quelle erhält dadurch eine eigene Priorität, sodass genau geregelt ist, welcher Interrupt Vorrang hat. Dabei darf es natürlich auch gleiche Prioritäten geben, die folglich nacheinander abgearbeitet werden müssen. Gibt es beispielsweise zwei Schrittmotoren, kann das zu einem leichten Jittern der Schrittimpulse führen. Solche Überlegungen sind beim Systementwurf sehr wichtig. Letztlich ist die Initialisierung der Prioritäten jedoch mit einer Zeile getan.

```
// enable 16 interrupt priorities
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);

// relocate vector table to 0x80004000
NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x4000);
```

Listing 4.7 Initialisierung des NVI-Controllers

Im Listing steht eine weitere wichtige Zeile, die die Lage der Vektortabelle im FLASH-Speicher betrifft. Unser Betriebssystem beginnt im Speicher hinter dem Bereich für den Bootloader. Dieser Bereich ist 16 kB groß, weil das die Größe des ersten FLASH-Segments ist. Damit beginnt das Betriebssystem bei der Adresse 0x80004000. Solange der NVIC nichts davon erfährt, würde er Interrupts über die Standard-Vektortabelle bei 0x80000000 aufrufen wollen. Deshalb muss dem NVIC der Offset zur Vektortabelle des Betriebssystems bekannt gegeben werden, also die 0x4000. Damit sind die Interrupts grundsätzlich konfiguriert. Jede Interrupt-Quelle muss dann noch separat konfiguriert werden. Das schauen wir uns im Folgenden am Beispiel der externen Interrupts an.

4.1.7 UART-Interfaces

Mit jeder neuen Hardware, die wir bisher initialisiert haben, kam eine neue spezifische Konfiguration hinzu – so auch jetzt wieder bei der Initialisierung einer seriellen Schnittstelle. Es geht um die UART2, das ist die Schnittstelle, über die wir auch schon den Download eines Programms im Bootloader realisiert haben. Die Besonderheit ist jetzt die Konfiguration eines *Interrupt-Vektors*. Während beim Bootloader die UART-Schnittstelle für den Download lediglich im *Polling Mode* betrieben wurde, sollen die drei UARTs beim Betriebssystem ausschließlich im *Interrupt Mode* verwendet werden.

Zunächst sind aber wieder die bereits bekannten Schritte nötig – Konfiguration der Pins RXD0 und TXD0 als alternative Funktion sowie die Konfiguration der UART selbst. Diese beiden Schritte sind identisch mit der Konfiguration im Bootloader.

```
// --- Defines ---
// USART2 as RS232
// PA2  = TXD0
// PA3  = RXD0

#define UART_PINS            GPIOA
#define TXD0                 GPIO_Pin_2
#define RXD0                 GPIO_Pin_3

#define TXD0_SRC             GPIO_PinSource2
#define RXD0_SRC             GPIO_PinSource3

// --- Init UART2 as RS232 ---
VOID InitRS232()
{
    USART_InitTypeDef uart_init = {0};
    GPIO_InitTypeDef  gpio_init = {0};
    NVIC_InitTypeDef  nvic_init = {0};

    // config alternate function
    GPIO_PinAFConfig(UART_PINS, TXD0_SRC, GPIO_AF_USART2);
    GPIO_PinAFConfig(UART_PINS, RXD0_SRC, GPIO_AF_USART2);

    // config TXD0, RXD0
    gpio_init.GPIO_Pin   = TXD0 | RXD0;
    gpio_init.GPIO_Mode  = GPIO_Mode_AF;
    gpio_init.GPIO_OType = GPIO_OType_PP;
    GPIO_Init(UART_PINS, &gpio_init);

    // UART data format setup 115200,8N1
    uart_init.USART_Mode       = USART_Mode_Rx | USART_Mode_Tx;
    uart_init.USART_WordLength = USART_WordLength_8b;
    uart_init.USART_StopBits   = USART_StopBits_1;
    uart_init.USART_Parity     = USART_Parity_No;
    uart_init.USART_BaudRate   = 115200;
    USART_Init(USART2, &uart_init);
```

```
    // UART Interrupt Vector
    nvic_init.NVIC_IRQChannelPreemptionPriority = 9;
    nvic_init.NVIC_IRQChannel = USART2_IRQn;
    nvic_init.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&nvic_init);

    // enable receive interrupt
    USART_ITConfig(USART2, USART_IT_RXNE, ENABLE);

    // enable finally
    USART_Cmd(USART2, ENABLE);
}

// --- USART2 interrupt handler ---
VOID USART2_IRQHandler()
{
    if (USART_GetITStatus(USART2, USART_IT_RXNE) == SET)
    {
        // RX buffer not empty
    }

    if (USART_GetITStatus(USART2, USART_IT_TXE) == SET)
    {
        // TX buffer empty
    }
}
```

Listing 4.8 Initialisierung einer interruptfähigen UART

Neu ist jetzt die Einrichtung und Anmeldung eines UART-Receive-Interrupts beim *Nested Vectored Interrupt Controller* (NVIC). Analog zu den bereits bekannten Strukturen für die Initialisierung von Hardwaremodulen gibt es dafür die Struktur `NVIC_InitTypeDef`.

Der primäre Parameter in der Struktur `NVIC_InitTypeDef` ist die Kanalnummer `NVIC_IRQChannel`. Die Kanalnummer korrespondiert mit einem Index in der Interrupt-Vektortabelle. Die Indizes darin sind fest in der Hardware hinterlegt – man kann also nicht selbst bestimmen, welchen Index man verwenden möchte. Am Beispiel unseres Moduls UART2 werden alle auftretenden Interrupts an die Funktion weitergeleitet, deren Adresse an Index 38 in der Vektortabelle steht. Natürlich muss man sich diese Nummern weder merken, noch muss man sie überhaupt kennen. Die CMSIS-Library hat hierfür bereits Konstanten vordefiniert, und in unserem Fall wäre das `USART2_IRQn`.

Der zweite wichtige Parameter ist die Priorität des Interrupts, die mit dem Wert in `IRQ-ChannelPreemptionPriority` festgelegt wird. Wie im Kapitel zuvor beschrieben, unterscheidet der NVI-Controller in Preemption- und in Sub-Priorities. Man kann in der Struktur daher auch beide Werte definieren. Da wir aber den NVI-Controller nur auf Preemption-Prioritäten konfiguriert haben, definieren wir auch nur diesen einen Wert. Der Wertebereich liegt zwischen 0 und 15, wobei die höchste Priorität die Nummer 0 hat! Unsere UART bekommt eine mittlere, eher etwas niedrigere Priorität zugewiesen.

Bei gesetztem `ENABLE`-Flag wird mit dem Aufruf der Funktion `NVIC_Init()` der Interrupt-Kanal aktiviert. Das heißt nicht, dass nun schon Interrupts ausgelöst werden. Es besteht lediglich eine Verbindung zwischen möglichen Interrupt-Quellen der UART mit dem *Interrupt-Handler*. Woher weiß der NVIC nun aber, welches der Interrupt-Handler ist?

Bei der Definition der Vektortabelle in der Datei *startup_stm32f4xx.c* werden bereits alle möglichen Interrupt-Handler namentlich vordefiniert. Der Interrupt-Handler an Index 38 der Tabelle, dem Index unseres UART2-Moduls, hat den Namen `USART2_IRQHandler()`. Nun kann C keine Adresse auf eine Funktion vergeben, wenn es diese Funktion gar nicht gibt, und da wir nur die wenigsten Interrupt-Handler tatsächlich definieren, könnte der C-Compiler die Vektortabelle gar nicht vollständig ausfüllen. Hier hilft ein Attribut – man kann auch sagen, eine vendorspezifische Compiler-Erweiterung – mit der Bezeichnung `__attribute__(weak)`. Es lohnt sich, diese Konstruktion einmal genauer anzuschauen, denn im Grunde ist es genau der gleiche Mechanismus, wie ihn objektorientierte Sprachen für das Überschreiben von *virtuellen Methoden* verwenden.

Allen namentlich vordefinierten Interrupt-Handlern wird ein *Default-Handler* zugewiesen, in dem Fall eine leere Funktion. Damit kann der C-Compiler die Vektortabelle zunächst initialisieren. Falls der Linker aber irgendwo eine Funktion mit dem gleichen Namen findet, ersetzt er den Eintrag in der Vektortabelle mit der Adresse dieser neuen Funktion. Man kann an der Stelle auch sagen, dass die neue Funktion die Default-Implementierung überschreibt.

Da wir einen eigenen Interrupt-Handler definieren, ersetzt der Linker den zugehörigen Eintrag in der Vektortabelle. Diese implizite Zuordnung zwischen einem Kanal des NVIC und dem passenden Interrupt-Handler ist allerdings schon etwas verwirrend, zumal es auch noch einer Compiler-Erweiterung bedurfte, um sie zu ermöglichen. Hier darf man durchaus kritisch fragen, ob es dafür nicht eine bessere Lösung gegeben hätte.

Ein Detail zum Interrupt-Handler muss man noch beachten. Wir haben nun einen Interrupt-Handler, der nach abgeschlossener Initialisierung für jedes empfangene Zeichen aufgerufen wird. Genau genommen kann der Handler den Grund seines Aufrufs jedoch gar nicht wissen, denn alle möglichen Interrupts der UART müssen von diesem einen

Handler bearbeitet werden. Zunächst ist das nur der *Receive-Interrupt*, aber wir werden sehen, dass zumindest noch der *Transmit-Interrupt* hinzukommt. Daher muss ein Interrupt-Handler gegebenenfalls auf mehrere mögliche *Interrupt-Status-Flags* prüfen und alle behandeln. In Listing 4.8 ist das bereits angedeutet, den genauen Inhalt der Funktion besprechen wir in Abschnitt 4.2.7, »Serielle Datenübertragung«, bei der Implementierung der seriellen Funktionen.

4.2 Abstraktionen

Bisher haben wir exemplarisch einige Hardwaremodule in unserem STM32 initialisiert bzw. konfiguriert. Dazu zählen die I/O-Pins, die Analogeingänge und natürlich unser SPI-Systembus. Dabei konnten wir komplett auf der CSMIS-Library aufbauen. Die Library lieferte uns alle Funktionen, die wir für diesen ersten Schritt benötigten. In diesem Abschnitt soll es nun darum gehen, Hardwarezugriffe zu abstrahieren. Was heißt das?

Natürlich bietet die CMSIS-Library auch Funktionen an, um etwa auf GPIO-Pins zuzugreifen oder Analogwerte einzulesen. Wir wollen jedoch einen eigenen *Hardware Abstraction Layer* (HAL) erstellen. Die Funktionen der CMSIS-Library sind deshalb nicht schlecht oder gar ungeeignet, aber sie sind nicht optimal, wenn es um eine spezielle Hardware geht. So kann die CMSIS-Library beispielsweise nicht wissen, wie wir unseren AD-Wandler verwenden, ob wir vielleicht eine externe oder die interne Referenz nutzen. Selbst der einfache Zugriff auf I/O-Pins lässt sich eleganter realisieren, wenn wir wissen, wie die Hardware aufgebaut ist.

Mit einem Hardware Abstraction Layer schafft man sich im Grunde eine optimierte Schnittstelle, die mittels Funktionen die verfügbare Hardware zum Betriebssystem hin kapselt. Man könnte umgekehrt auch sagen, dass man die Hardware vor dem Betriebssystem verbirgt. Der wesentliche Vorteil ist dabei nicht allein die optimale Implementierung einer solchen Schnittstelle, es ist die Schnittstelle selbst. Wenn man das Betriebssystem auf eine andere Hardware portieren möchte, muss man den HAL neu implementieren oder anpassen, alle Aufrufe in den HAL bleiben aber unverändert. Unser HAL ist eine Zwischenschicht zwischen der CMSIS-Library und dem Betriebssystem.

4.2.1 Port-Zugriffe

Bei der Initialisierung der GPIOs mithilfe der Funktion `GPIO_Init()` wurde als erster Parameter immer der Port, also etwa `GPIOA`, `GPIOB` etc., angegeben. Im Grunde erwartete die Funktion `GPIO_Init()` aber einen Zeiger auf eine Datenstruktur vom Typ `GPIO_TypeDef`.

Kapitel 5
Embedded BASIC

Auch wenn manche C für die beste Programmiersprache halten, gibt es »die beste Programmiersprache« nicht! Es gibt vielleicht eine, die am besten geeignet ist.

Bisher haben wir eine Hardware und ein dazu passendes Betriebssystem erstellt. Jetzt sollen damit auch Anwendungen programmiert werden. Warum aber nun noch eine weitere Programmiersprache, wir konnten doch bisher alle Aufgaben auch gut mit C lösen?

5.1 Ansatz

In der Regel gelten für die Betriebssystem- und für die Anwendungsentwicklung unterschiedliche Anforderungen. In der Praxis werden die Aufgaben meist auch verschiedenen Personen oder sogar verschiedenen Abteilungen zugeordnet. Während man beim Betriebssystem sehr auf exaktes Timing, hohe Effizienz und auf direkte Hardwarezugriffe Wert legt, kommt es bei der Anwendungsentwicklung doch eher auf Zuverlässigkeit, Toleranz und auf die komfortable Nutzung vorgefertigter Funktionen an. Beide Bereiche haben ganz unterschiedliche Abstraktionsniveaus. Am Beispiel etwa einer zeitgesteuerten Bewässerung benötigt man in der Applikation die Zeit und die Pumpe als Variablen, sodass man komfortabel und leicht verständlich schreiben kann: IF Time == 10:30 THEN Pump = high.

Während die Entwicklung des Betriebssystems zumeist an einem Arbeitsplatz oder im Labor unter Zuhilfenahme von Messgeräten wie Oszilloskopen oder auch einem In-System-Debugger erfolgt, entsteht die Anwendungssoftware dagegen oft vor Ort an einer Anlage. Die Applikation entsteht in kleinen Schritten, wobei jeder einzeln implementiert und getestet werden muss. Dabei sind insbesondere kurze Testzyklen wünschenswert, die Compile- und Download-Zeiten müssen daher möglichst kurz sein.

Die Lesbarkeit bzw. Verständlichkeit eines Anwendungsprogramms ist ein nicht zu vernachlässigender Aspekt. In der Praxis kann eine Anwendung über Jahre entwickelt wer-

den, und zwischen einzelnen Entwicklungsstufen vergeht sehr viel Zeit. An einer bestehenden Applikation nach langer Zeit Änderungen vorzunehmen, gelingt nur, wenn der Quelltext verständlich ist. Das hängt natürlich in erster Linie an der Größe der Applikation sowie an der Art und Weise, wie das Programm geschrieben und dokumentiert wurde – eine klare Syntax und die Möglichkeit zur Kapselung helfen dabei jedoch ebenfalls sehr.

Nicht zuletzt sind Toleranz und eine gewisse Sicherheit, die eine Applikationssprache mit sich bringt, sehr nützlich. Bei C als Systemsprache steht dagegen die Performance an oberster Stelle, was dort natürlich richtig ist. Um aber bei dem Beispiel der zeitgesteuerten Pumpe zu bleiben – da ist die Performance zweitrangig.

Bei genauem Vergleich sieht man, dass eine Applikationssprache deutliche Vorteile gegenüber C haben kann. Das erkauft man sich in der Regel zwar mit einer geringeren Performance, die aber teilweise durch vorgefertigte Funktionalitäten kompensiert werden kann. In Summe überwiegen dann die Vorteile.

5.1.1 Skriptsprache

Die Schnittstellen, die ein Betriebssystem anbietet, etwa um einen Text über eine serielle Schnittstelle auszugeben, werden meist über sogenannte *Application Programming Interfaces* (APIs) abgebildet. Das sind zahlreiche Funktionen, die in verschiedenen Gruppen zusammengefasst werden. Am Beispiel einer seriellen Ein- und Ausgabe wären das etwa `PutStringSerial()` und `GetStringSerial()`.

Die Idee ist nun, anstelle einer klassischen API eine Skriptsprache anzubieten. Im Grunde sind die Kommandos der Skriptsprache dann nichts anderes als die Funktionsaufrufe einer API-Schnittstelle, nur eben komfortabler. Eine einfache Textausgabe sähe als API-Aufruf in C etwa folgendermaßen aus:

```
PutStringSerial("Hello World!");
```

als Skriptkommando dagegen so:

```
print "Hello World!".
```

Der wirkliche Vorteil der Skriptkommandos wird allerdings erst ersichtlich, wenn man eine etwas kompliziertere Ausgabe anschaut, wie etwa bei einer Berechnung mit Formatierung und anschließender Zeilenschaltung:

```
print "Ergebnis = ", sin(45), cr, nl
```

Dafür müsste man in C zunächst eine Textvariable deklarieren und das Ergebnis als String in diese Variable formatieren. Die Hilfsvariable könnte man dann mittels der API-

Funktion `PutStringSerial()` ausgeben. Das sind aber immerhin schon drei Zeilen und eine temporäre Variable:

```
char txt[20] = {0};

sprintf(txt, "Ergebnis = %f\r\n", sin(45));
PutStringSerial(txt);
```

Eine Sprache anstelle von API-Funktionen kann also Zwischenwerte berechnen und zudem eine beliebige Anzahl an Parametern zulassen, wofür man sonst mehrere API-Funktionen aufrufen müsste. In C gibt es zwar das Konzept der variadischen Funktionen (*Variadic Functions*) (siehe Quelle 11 im Literaturverzeichnis), die aber leider keine Typinformationen unterstützen.

Das sind jedoch längst nicht alle Vorteile, denn auch den Zugriff auf die Hardware kann man deutlich komfortabler gestalten, als das mit C-Funktionen möglich ist. Der Zugriff auf einen Port des Controllers würde beispielsweise über eine Variable erfolgen, die den Port repräsentiert – also etwa `Port0`, `Port1` etc. Einen einzelnen Pin eines Ports würde man dann über `Port0.0`, `Port0.1` etc. erreichen. Eine logische Verknüpfung mehrerer Pins wäre so mit einer leicht verständlichen Syntax möglich:

```
Port0.1 = Port1.2 or Port1.3
```

wobei hier zwei Pins gelesen und logisch verknüpft würden und das Ergebnis an Pin 1 von `Port0` ausgegeben werden würde. Mit dem Lese- und Schreibzugriff solcher Hardwarevariablen sind jeweils passende Zugriffsfunktionen im Betriebssystem verbunden – letztlich sind es also schon Funktionsaufrufe, die aber in einer geschickten Syntax verpackt werden. In der objektorientierten Programmierung nennt man solche Variablen auch *Properties*.

Der Vergleich von Library-Funktionen mit einer Skriptsprache scheint auf den ersten Blick schon etwas gewagt. Werden da nicht verschiedene Dinge miteinander verglichen? Nein, die Bezeichnung API steht grundsätzlich für eine Schnittstelle zur Programmierung von Anwendungen, und die kann klassisch in Form von Library-Aufrufen realisiert werden, aber auch in Form eines Datenprotokolls oder etwa über Gerätedateien – wie unter Linux – oder eben als Skriptsprache.

Die Nutzung einer Skriptsprache ist dabei, gerade im Zusammenhang mit einer Mikrocontroller-Hardware zur Steuerung von Abläufen, sehr naheliegend. Für eine vollwertige, sprich *Turing-vollständige* Skriptsprache braucht man allerdings noch etwas mehr – mindestens noch Variablen, Bedingungen und Verzweigungen. In den kommenden Abschnitten wollen wir alle wesentlichen Sprachelemente von Embedded BASIC kennenlernen, aber beginnen wir zunächst mit einem Gedankenexperiment.

5.1.2 Möglichkeiten

Eine Skriptsprache für unser Betriebssystem anstelle einer klassischen API eröffnet uns eine ganz neue Möglichkeit. Dabei wird die Skriptsprache gleichzeitig zur Programmierung als auch zur Konfiguration genutzt. Schauen wir dazu folgendes Beispiel an.

Angenommen, Sie wollten eine mehrkanalige Schaltuhr für die Bewässerung Ihres Gartens entwickeln, die natürlich über eine Handy-App programmierbar sein soll. Mit der App wollen Sie die Schaltzeiten komfortabel eingeben und verwalten. Jede Schaltzeit ist ein Datensatz, der aus einem oder mehreren Wochentagen, einer Einschalt- oder einer Ausschaltzeit besteht sowie aus einem zu schaltenden Kanal. Die Kommunikation zwischen Handy und der Schaltuhr soll beispielsweise über Bluetooth erfolgen. Die Hardware dafür ist denkbar einfach – unser Breakout-Board, ein Bluetooth-Modul (z. B. das HC-06 oder das HM-10) an RXD0 und RXD1 sowie ein paar Relais für die Ausgänge – fertig!

Umgang mit 230-V-Netzspannung!

Auch wenn die Schaltungstechnik recht einfach ist, der Umgang mit 230-V-Netzspannung bleibt dem Fachmann vorbehalten! Die weitverbreiteten kleinen und günstigen (meist blauen) Relais-Module sind nicht für das Schalten von Netzspannung geeignet und dürfen auf keinen Fall dafür verwendet werden. Bei Bedarf immer einen Elektriker hinzuziehen!

Überlegen Sie bitte kurz einmal, wie Sie eine derartige Schaltuhr unter Nutzung einer klassischen API-Schnittstelle und mittels Programmierung in C auf einem Mikrocontroller realisieren würden.

Klar ist sicher, dass es eine Liste aller Datensätze geben muss, die aus der Handy-App an die Schaltuhr übertragen werden muss. Die Übertragung entspricht in dem Fall einer seriellen Verbindung, wofür ein einfaches Protokoll benötigt wird. Das C-Programm iteriert dann zyklisch über alle Datensätze in dieser Liste und vergleicht sie jeweils mit der internen Uhrzeit. Dazu bedarf es noch einer Möglichkeit, die interne Uhrzeit zu stellen oder mit dem Handy zu synchronisieren. Das ist technisch alles gar nicht so kompliziert, aber es muss erst einmal programmiert werden. Außerdem ergeben sich im Verlauf der Entwicklung mit großer Wahrscheinlichkeit die folgenden Detailfragen:

- Wie werden die Datensätze gespeichert?
- Wie viele Datensätze soll das Programm vorhalten?
- Welche weiteren Optionen oder neuen Features sind denkbar?
- Wie erfolgt die Versionierung, wenn sich die Datensätze ändern?
- usw.

Im Grunde hat bestimmt jeder Programmierer, der sich mit eingebetteten Systemen und Mikrocontrollern beschäftigt, schon einmal eine vergleichbare Aufgabe bearbeitet und sieht die Lösung quasi schon vor seinem inneren Auge.

Nun stellt sich die Frage, wie man diese Anwendung unter Nutzung einer Skriptsprache anstelle der API besser realisieren könnte, und die verblüffende Antwort lautet: Das ist gar nicht nötig!

Das einzige zu erstellende Programm ist nämlich die Handy-App, die ohnehin für die Eingabe und die Speicherung der Schaltzeiten benötigt wird. Diese App generiert aus den Schaltzeiten direkt ein Skriptprogramm für unseren Mikrocontroller und überträgt das gesamte Programm mittels des Bootloaders in den Mikrocontroller – fertig!

Die Handy-App und das daraus generierte Skriptprogramm könnten im einfachsten Fall etwa so aussehen:

Abbildung 5.1 Ansicht der Handy-App für die Schaltzeiten

```
loop
    if Day == MONDAY then

        if Time ==  6:00 then Port1.0 = high
        if Time ==  7:30 then Port1.0 = low
        if Time == 21:45 then Port1.1 = high

    elseif Day == TUESDAY then

        ...

    endif
endloop
```

Listing 5.1 Auszug aus dem aus der Handy-App generierten Skriptprogramm

Nun wäre eine reine Schaltuhr wirklich nicht mehr zeitgemäß. Vielmehr würde man bei einer Beregnungsautomatik auch noch die tatsächliche Bodenfeuchte oder zumindest die aktuelle Temperatur berücksichtigen wollen. Dafür könnte man dann Regeln mit Grenzwerten definieren, die man mit in der App hinterlegen kann. Letztlich werden daraus weitere Bedingungen im generierten Skript – das Prinzip bleibt dabei unverändert. Das sollte lediglich ein einfaches Beispiel sein, um die Flexibilität einer Skriptsprache zu verdeutlichen. Die App gibt es so leider noch nicht.

Analysieren wir die Entwicklungsaufgabe aber noch etwas genauer. Es gibt hier zwei Domänen: auf der einen Seite die Erstellung einer Handy-App und auf der anderen Seite die hardwarenahe Programmierung des Mikrocontrollers. Beide Domänen erfordern spezielles Wissen, und wenige Softwareentwicklerinnen und -entwickler kennen sich auf beiden Gebieten gleichermaßen gut aus. Außerdem bedarf es einer engen Abstimmung zwischen den Gebieten, weil durch die zu übertragenden Daten eine Abhängigkeit zwischen ihnen besteht.

Durch die Skriptsprache entfällt jedoch die Entwicklung eines speziellen C-Programms für den Mikrocontroller in der Schaltuhr, und damit entfällt auch eine der beiden Domänen. Der Entwickler der Handy-App kann aus den Schaltzeiten und Regeln einen Quelltext generieren und diesen per Bluetooth übertragen. Somit wird die Skriptsprache gleichsam zu einer Art Konfigurationssprache, in der sowohl der Ablauf als auch die Zeiten definiert werden.

Wäre das nicht auch mit C möglich? Ob die Handy-App nun den Quelltext einer Skriptsprache oder den Quelltext in C erstellt, wäre sicher kein nennenswerter Unterschied.

Allerdings wäre es dann kaum möglich, das C-Programm auf dem Handy zu kompilieren, und außerdem müsste der kompilierte Binärcode anschließend noch mit dem OS auf dem Mikrocontroller gelinkt werden. In jedem Fall wird der Aufwand deutlich größer. An dieser Stelle ist die Verwendung einer interpretierten Skriptsprache merklich einfacher.

5.2 Sprache

Wodurch unterscheiden sich denn Skriptsprachen eigentlich von klassischen Programmiersprachen? Stark vereinfacht, könnte man vielleicht sagen, dass Skriptsprachen interpretativ ausgeführt werden, während andere Programmiersprachen einen Compiler verwenden. Das ist allerdings nicht ganz eindeutig, denn viele moderne Skriptsprachen kompilieren den Quelltext unmittelbar vor der Ausführung noch auf dem Zielsystem. Man kann aber schon feststellen, dass die Weitergabe eines Skriptprogramms praktisch immer in Form des Quelltexts erfolgt, wohingegen kompilierte Programme in einer ausführbaren Binärform, etwa als *Executable* (**.exe*), weitergegeben werden.

Darüber hinaus sind Skriptsprachen in der Regel syntaktisch deutlich einfacher aufgebaut und haben oft auch eine weniger strenge Typisierung. Aber auch hier gibt es Unterschiede, und eine pauschale Kategorisierung ist nicht so einfach möglich.

In gewisser Weise sind Skriptsprachen aber eher kleinere Sprachen, wobei »kleiner« hier im Sinne eines geringeren Entwicklungsaufwands zu verstehen sein soll. Die Komplexität eines C-Compilers ergibt sich nämlich nicht aus der C-Syntax, sondern aus dem Anspruch einer bestmöglichen Optimierung. Da Skriptsprachen meist nur wenig optimieren, ist deren Entwicklung deutlich einfacher.

In den nachfolgenden Abschnitten soll es nun um die Definition und um eine eingehende Beschreibung einer einfachen, aber trotzdem sehr praktikablen Skriptsprache für unser Single-Chip-System gehen. Natürlich haben alle Programmiersprachen einen treffenden Namen, und diese soll »Embedded BASIC« heißen. Die Bezeichnung ist eine Anlehnung und zugleich eine Hommage an das BASIC, das bereits 1964 von John G. Kemeny und Thomas E. Kurtz entwickelt wurde.

5.2.1 Programmiermodell

Bei Embedded BASIC handelt es sich primär um eine prozedurale Sprache. Sie unterstützt allerdings auch Grundkonzepte der Objektorientierung sowie ein einfaches Modulkonzept. Außerdem unterstützt die Sprache präemptives Multitasking. Letzteres ist gerade für Steuerungsaufgaben sehr nützlich. Schauen wir dazu folgendes Beispiel an.

```
import Target

// names
symbol Pump = Out1
symbol Heater = Out2
symbol TempSensor = AIN1

// control tasks
task TempControl, 0.1, PRIORITY_HIGH
task PumpControl, 0.5, PRIORITY_LOW

// temp control task
function TempControl()
float Temp = TempSensor * 105.0 / 10000 - 25

    if Temp < 22.5 then Heater = high
    if Temp > 23.5 then Heater = low
endfunction

// pump control task
function PumpControl()

    select Day
        case MONDAY to FRIDAY
            if Time ==  8:30 then Pump = high
            if Time ==  9:00 then Pump = low
        case else
            if Time == 10:30 then Pump = high
            if Time == 11:00 then Pump = low
     endselect
endfunction

// idle loop
loop

endloop
```

Listing 5.2 Beispiel in Embedded BASIC

Im obigen Programm werden zwei Tasks gestartet. Der erste Task hat den Namen `TempControl` und soll einen einfachen Zweipunktregler realisieren. Der zweite Task mit dem

Namen `PumpControl` soll eine Pumpe zeitabhängig ein- bzw. ausschalten. Jeder Task benötigt eine Funktion mit dem gleichen Namen wie der Task, die dann zyklisch aufgerufen wird. Die Zykluszeit wird beim Start des Tasks als zweiter Parameter nach dem Namen angegeben, d. h., die Funktion `TempControl` wird alle 100 ms aufgerufen, während `PumpControl` nur alle 500 ms aufgerufen wird. Aufgrund seiner Priorität unterbricht ein Task mit hoher Priorität einen eventuell gerade laufenden Task mit niedriger Priorität. Dieses Prinzip heißt *präemptiv* und sorgt für kurze Reaktionszeiten und damit für Echtzeitverhalten.

Immer dann, wenn kein Task ausgeführt wird, läuft der sogenannte *Idle-Task* – das sind die Anweisungen, die zwischen LOOP und ENDLOOP steht. Obwohl im obigen Beispiel der Idle-Task leer ist, eignet sich dieser Teil besonders für Kommunikations- und Überwachungsfunktionen, weil dieser Task kontinuierlich durchläuft. Der Idle-Task hat dafür auch die niedrigste Priorität und wird von allen Tasks unterbrochen.

Durch die Kombination aus bis zu 16 zyklischen Echtzeit-Tasks und einem kontinuierlichen Idle-Task stellt unsere Skriptsprache ein sehr flexibles Programmiermodell zur Verfügung, das insbesondere für die Realisierung von kleinen Steuerungsaufgaben sehr gut geeignet ist.

5.2.2 Programmstruktur

Alle Programmiersprachen erfordern, dass Programme einer prinzipiellen Strukturierung folgen. In Abschnitt 7.2 über die Syntaxanalyse wird das noch genauer erläutert werden, es ergibt sich direkt aus der Grammatik, die einer Sprache zugrunde liegt.

In unserer Skriptsprache besteht ein Programm daher aus einer Liste von Anweisungen, d. h., die Ausführung beginnt mit der Anweisung in der ersten Zeile und endet mit der Anweisung in der letzten Zeile. Das ist grundlegend anders als etwa bei einem C-Programm, das aus einer Liste von Funktionen besteht. Die Ausführung beginnt mit dem Aufruf der `main()`-Funktion und endet auch, wenn die Funktion endet. Diesen Unterschied wollen wir noch genauer betrachten, weil unsere Skriptsprache daraus wesentliche Vorteile ziehen kann.

Ein Skriptprogramm besteht aus mindestens einer Quelldatei. In der Regel werden jedoch gleich zu Beginn weitere Dateien, sogenannte Module, importiert. Im Beispiel aus Listing 5.2 wird nur das `Target`-Modul importiert – in der Praxis sind es aber meist mehrere Module. Im `Target`-Modul, das immer zuerst importiert werden sollte, werden grundlegende Deklarationen über die Zielhardware zusammengefasst. Dazu zählen Informationen wie etwa der verfügbare Speicher für Programm und Daten, aber auch schon einige Definitionen generischer Namen für Ein- und Ausgänge. Stellen Sie sich

eine Importanweisung vor, als würde der gesamte Quelltext des importierten Moduls an der Stelle der Importanweisung eingefügt.

Da die Ausführung nun in der ersten Zeile beginnt, werden also zuerst die Anweisungen der importierten Module ausgeführt. Module können selbst auch wieder Module importieren, sodass immer zuerst die Anweisungen aller importierten Module ausgeführt werden. Auf diese Weise können sich Module zunächst selbst initialisieren, bevor sie dann verwendet werden. Sie können eigene Variablen und Instanzen anlegen, Funktionen aufrufen und gegebenenfalls auch schon Tasks starten. Mit den Modulen kann auch gezielt Hardware initialisiert werden. Im Vergleich dazu kann ein C-Programm zwar Module mittels der Include-Anweisung einbinden, diese Module können aber erst aus der `main()`-Funktion heraus initialisiert werden. Vereinfacht könnte man auch sagen, dass die Module in Embedded BASIC eigenständiger sind.

Die Deklarationen von Variablen müssen in dem Zusammenhang und zum besseren Verständnis auch als Anweisungen angesehen werden. Es ist guter Programmierstil, wenn Variablen direkt bei ihrer Deklaration einen Initialwert zugewiesen bekommen. Diese Zuweisung ist nur deshalb möglich, weil eine Deklaration quasi auch eine Zuweisung darstellt. Der Skript-Interpreter führt bei einer einfachen Variablendeklaration wie dieser:

```
int i = 0
```

folgende Anweisungen aus: Trage in die Variablenliste für i den Typ Integer ein, reserviere dafür 4 Byte im Speicher und initialisiere diese mit 0. Das ist auch deshalb von Bedeutung, weil die Variable `i` erst dann verwendet werden kann, wenn sie deklariert wurde. Deklarationen sollten daher zu Beginn eines jeden Moduls stehen – gleich im Anschluss an die Importanweisungen.

Das lässt sich analog auch auf Funktionen und sogar auf Klassen übertragen. Auch strukturierte Deklarationen sind so gesehen nur Anweisungen im Sinne der Programmstruktur. Während Funktionen nur einmalig mit ihrer Adresse registriert werden, generieren Klassen einen Prototyp im Speicher, der dann bei jeder Instanziierung verwendet wird.

Damit sollte eine Skriptdatei in Embedded BASIC also immer etwa folgende Struktur haben, wobei die konkrete Ausgestaltung und Reihenfolge der einzelnen Elemente weitestgehend frei wählbar bleibt.

Das Hauptmodul besitzt am Ende noch eine *Main-Loop*. Diese Main-Loop ist der *Idle-Task*, der immer dann ausgeführt wird, wenn gerade kein Echtzeit-Task läuft. Wenn die Main-Loop beendet wird, beendet sich das Programm. Möchte man die gesamte Applikation also über ein Kommando beenden und neu starten, kann die Main-Loop einfach ein Abbruchkriterium enthalten.

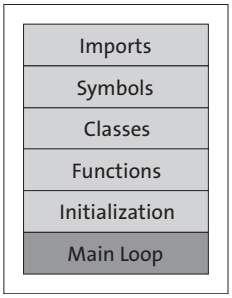

Abbildung 5.2 Grundstruktur eines Skriptmoduls

Zur Sicherheit sollte man im Idle-Task auch immer das Watchdog-Kommando aufrufen. Sollte aus irgendeinem Grund ein Task »hängen bleiben« und wird demzufolge der Idle-Task nicht mehr ausgeführt, beendet der Watchdog die Anwendung nach etwa vier Sekunden.

Obwohl die Art und Weise der Ausführung von Skripten in Embedded BASIC von der anderer, insbesondere kompilierter Programmiersprachen abweicht, ist sie doch eher intuitiver. Ab Version 9 unterstützt beispielsweise auch C# Anweisungen auf der obersten Programmebene – also ohne die Notwendigkeit einer *statischen* `main()`-Funktion. Das entspricht in etwa der Art der Programmstrukturierung, wie wir sie in unserem Skript verwenden.

5.2.3 Kontrollstrukturen

Man könnte meinen, dass es mit Blick auf die Kontrollstrukturen kaum nennenswerte Unterschiede zwischen verschiedenen Programmiersprachen gibt. Alle Programmiersprachen bieten Entscheidungen an, meist in Form einer IF-Anweisung, und sie unterstützen Schleifen, wie etwa eine WHILE- oder eine FOR-Schleife. Gibt es da wirkliche Neuerungen bei Kontrollstrukturen? Bei genauerem Hinsehen findet man schon innovative Konstrukte, an die man sich als Programmierer sehr schnell gewöhnt. Beispiele dafür sind etwa die Range-Based-Loops, wie sie in Python oder auch in C++ unterstützt werden. Besonders elegant ist die Unterstützung der Foreach-Loops in C#. Java unterstützt ein ähnliches Verhalten als Bestandteil der For-Schleife. Das sind mächtige, aber technisch leider auch kompliziert zu implementierende Kontrollstrukturen.

Kontrollstrukturen sind ein ganz wesentlicher Aspekt, und es lohnt sich, darüber genauer nachzudenken. Die am häufigsten verwendete Kontrollstruktur ist die IF-Anweisung. In Embedded BASIC kann die IF-Anweisung in verschiedenen Varianten geschrieben werden, damit die komfortabelste Schreibweise verwendet werden kann.

IF-Anweisung

Zunächst gibt es die klassische Variante, die zwischen IF und ENDIF einen Anweisungsblock bildet, der ausgeführt wird, wenn die Bedingung nach dem IF einen Wert von ungleich null ergibt. Bedingungen, in denen boolesche Ausdrücke wie etwa Vergleiche der Form `i < 10` vorkommen, werden dabei so interpretiert, dass FALSE == 0 und TRUE == 1 ergibt. Zur Auswertung von Ausdrücken kommen wir noch genauer, aber das unterscheidet unsere Skriptsprache vom ursprünglichen BASIC. Selbstverständlich gibt es auch noch eine IF-ELSE-Variante und darüber hinaus sogar eine IF-ELSEIF-Variante:

```
if i < 10 then
    <Anweisungen>
endif

if i < 10 then
    <Anweisungen>
else
    <Anweisungen>
endif

if i < 10 then
    <Anweisungen>
elseif i < 20 then
    <Anweisungen>
else
    <Anweisungen>
endif
```

Listing 5.3 If-Anweisungen mit Anweisungsblöcken

In einer IF-Anweisung können natürlich beliebig viele ELSEIF-Anweisungen nacheinander aufgelistet werden. Die Abarbeitung erfolgt dann exakt in der Reihenfolge, in der sie aufgeführt werden.

In der Praxis gibt es jedoch oft Fälle, bei denen nur einzelne Anweisungen zwischen IF und ENDIF stehen. Dafür wurde die IF-Anweisung noch um eine Kurzform ergänzt. So ist es möglich, eine oder auch mehrere Anweisungen direkt nach dem THEN in derselben Zeile zu schreiben. In dem Fall handelt es sich um eine Kurzschreibweise ohne abschließendes ENDIF.

```
if i > 10 then <Anweisungen>
if i > 10 goto <Label>
```

```
if i > 10 exit
if i > 10 return
```

Listing 5.4 Kurzformen der If-Anweisung

Darüber hinaus gibt es auch noch spezielle Kurzformen in Kombination mit anderen Anweisungen. Das sind die sogenannten bedingten Verzweigungen (*Conditional Branches*), wie IF-GOTO, IF-EXIT oder IF-RETURN. Allen Kurzformen ist jedoch gemein, dass es keinen ELSE-Zweig gibt.

SELECT-Anweisung

In der Tat könnte man bei einer IF-ELSEIF-Anweisung beliebig viele ELSEIF-Zweige angeben. Oft gibt es jedoch den Fall, dass man immer mit ein und demselben Anfangswert vergleicht. Das wäre dann einerseits unübersichtlich und andererseits auch noch uneffektiv. Für diese Fälle bietet sich die SELECT-Anweisung an. Mit ihr kann man einen Ausdruck auf verschiedene Werte oder sogar auf Wertebereiche prüfen und jeweils einen anderen Anweisungsblock durchlaufen.

```
select i
    case 1
        <Anweisungen>
    case 2, 3
        <Anweisungen>
    case 4 to 9
        <Anweisungen>
    case 10 to 13, 15 to 19
        <Anweisungen>
    case else
        <Anweisungen>
endselect
```

Listing 5.5 Select-Anweisung

In Embedded BASIC sind neben integralen, sprich ganzzahligen Werten auch rationale numerische Werte zulässig. Dabei muss man jedoch gegebenenfalls auf Rundungsfehler achten und die Vergleiche dann eher auf Bereiche als auf Einzelwerte auslegen.

Die Implementierung der SELECT-Anweisung testet den Ausdruck nach dem SELECT auf alle Werte bzw. Bereiche in der Reihenfolge, in der sie aufgeführt sind. Bei Übereinstimmung wird der zugehörige Anweisungsblock ausgeführt.

LOOP-Anweisung

Die Loop-Anweisung ist im Grunde eine Endlosschleife. Verwendung findet sie typischerweise beim Idle-Task, weil der kontinuierlich durchlaufen soll – immer nur unterbrochen durch die Echtzeit-Tasks. Eine Kombination aus der Loop-Anweisung und einem bedingten EXIT ergibt auch eine mitten- oder endgeprüfte Schleife.

```
loop
    <Anweisungen>
    if terminate exit
    <Anweisungen>
endloop
```

Listing 5.6 Loop-Anweisung, mittengeprüft

WHILE-Anweisung

Die klassische anfangsgeprüfte oder auch abweisende Schleife ist die WHILE-Schleife. Sie wird praktisch von jeder Programmiersprache in nahezu der gleichen Form unterstützt. Es gibt lediglich kleine syntaktische Unterschiede zwischen verschiedenen Sprachen. In Embedded BASIC ist die WHILE-Schleife ebenfalls fester Bestandteil.

```
while i
    <Anweisungen>
endwhile
```

Listing 5.7 While-Anweisung

FOR-Anweisung

Bei der FOR-Anweisung gibt es zwischen verschiedenen Sprachen deutliche Unterschiede. Das liegt insbesondere an der Art der Implementierung. Im Grunde gibt es immer drei Schritte, die eine FOR-Schleife ausmachen. Dazu gehören die Zuweisung des Anfangswerts, der Zählvorgang selbst und die Abbruchbedingung. In C-ähnlichen Sprachen werden diese drei Schritte jeweils separat angegeben. Dadurch ergibt sich eine sehr große Flexibilität.

In den meisten anderen Sprachen, wie auch in BASIC, ist die FOR-Schleife dagegen etwas spezifischer und intuitiver, dafür aber weniger flexibel. So werden explizit eine Zählvariable, ein Anfangs- und ein Endwert sowie optional auch noch eine Schrittweite angegeben. Wichtig ist, dass diese drei Werte gleich zu Beginn der Schleife einmalig berechnet werden. Der Zählvorgang sowie der Test auf das Abbruchkriterium erfolgen dann im Interpreter. Das macht die FOR-Schleife bei der Implementierung mit einem Interpreter

zwar effektiver, man kann aber beispielsweise kein laufendes Abbruchkriterium mehr formulieren. In der Praxis ist das aber meist keine Einschränkung, und wenn es nötig ist, kann man auch auf eine der anderen Schleifen ausweichen.

So gesehen ist die FOR-Schleife in Embedded BASIC die effektivste Variante für einen Zählvorgang, und neben den üblichen integralen Zählvariablen sind hier auch Gleitkommazahlen zulässig.

```
for i = 0 to 10
    <Anweisungen>
endfor

for i = 10 to 0 step -1
    <Anweisungen>
endfor
```

Listing 5.8 For-Anweisung mit Schrittweite

Im Beispiel oben ist auch gut zu erkennen, wie man vorwärts- und rückwärtszählende Schleifen erstellen kann.

EXIT-Anweisung

Jede der bisher besprochenen Schleifen kann mittels der EXIT-Anweisung zu jedem Zeitpunkt verlassen werden. EXIT bricht jedoch immer nur die innerste Schleife ab. Sofern also mehrere geschachtelte Schleifen verlassen werden sollen, muss man in jeder Schleife EXIT aufrufen. Alternativ kann man solche geschachtelten Schleifen auch in einer Funktion aufrufen und dort mittels RETURN die Funktion wieder verlassen.

Wichtig ist, dass man nicht mit GOTO in eine Schleife hinein- oder aus einer Schleife herausspringen darf! Das ist ein typisches Problem bei Interpretern. Kompilierte Sprachen haben damit in der Regel keine Problem. Der Grund dafür liegt in der Art der Implementierung von Schleifen. Ein Interpreter legt sich für eine Schleife einen Eintrag auf dem Stack an. Dieser Eintrag wird nur wieder entfernt, wenn das Schleifenende überlaufen wird. Bei einem Sprung würde es logischerweise zu einem Stackfehler kommen. In der Praxis ist das kein großes Problem, dennoch muss man es beachten.

Derartige *Misconceptions* gibt es wohl in jeder Programmiersprache. Dabei handelt es sich um eine Art mehrdeutige Implementierung, die leicht missverständlich werden kann. So besitzen etwa die Operatoren ++ und -- in allen C-ähnlichen Sprachen ein großes Potenzial für Fehler. In Python können Tippfehler zu impliziten Variablendeklarationen und damit zu Fehlverhalten führen. In Pascal gibt es das *Dangling-Else-Problem*

usw. Vielleicht machen ja kleine Fehler aber auch liebenswert, jedenfalls hat keine dieser *Misconceptions* je dazu geführt, dass eine Programmiersprache nicht mehr verwendet wurde.

GOTO-Anweisung

Obwohl der Informatiker Edsgar W. Dijkstra schon 1968 in seinem, inzwischen oft zitierten Zeitschriftenbeitrag *Go To Statement Considered Harmful* die Verwendung von GOTOs infrage stellte: »I became convinced that the go to statement should be abolished from all ›higher level‹ languages«, gibt es bis heute in fast jeder Programmiersprache noch eine GOTO-Anweisung. Es ist nicht sinnvoll, ein elementares Sprachkonstrukt pauschal abzulehnen, es ist aber sehr wohl berechtigt, über einen sinnvollen Umgang mit der GOTO-Anweisung nachzudenken. Es gibt Fälle, in denen man zwar auch ohne GOTO auskäme, die aber unter Verwendung von GOTOs eleganter gelöst werden können. In diesem Sinn ist die GOTO-Anweisung auch in Embedded BASIC enthalten.

ON-GOTO-Anweisung

Neben dem klassischen GOTO existiert in Embedded BASIC eine weitere Form – die ON-GOTO-Anweisung. Sie stellt eine einfache Variante dar, in Abhängigkeit von einem Index zu einem von mehreren Labels zu verzweigen.

```
    on i goto lab0, lab1, lab2
    goto ends

lab0:
    <Anweisungen>
     goto ends

lab1:
    <Anweisungen>
    goto ends

lab2:
    <Anweisungen>
    goto ends

ends:
```

Listing 5.9 On-Goto-Anweisung