Bearbeiten von „Testseite“ (Abschnitt)

Vorschau

Dies ist nur eine Vorschau. Die Seite wurde noch nicht gespeichert! → Zum Bearbeitungsfeld gehen

Inhaltsverzeichnis

Noch mehr Text

Hallo! tag

test6 Die Prozessorarchitektur ARM ist in allen Industrieanwendungen weit verbreitet und auch eine bedeutende Anzahl von Hobby- und Herstellerprojekten. In diesem Lernprogramm werden die Grundlagen der Programmierung von ARM-Prozessoren in Assemblersprache vermittelt.

Tutorial von Niklas Gürtler. Thread im Forum für Feedback und Fragen.

Einleitung

Warum Assembler?

Heutzutage gibt es eigentlich keinen Grund, Assemblersprache für ganze Projekte zu verwenden, da hochwertige Optimierungscompiler für Hochsprachen (insbesondere C und C ++) als kostenlose Open-Source-Software leicht verfügbar sind und die ARM-Architektur speziell für Hochsprachen optimiert ist Sprachen. Assembler-Kenntnisse sind jedoch nach wie vor hilfreich, um bestimmte Probleme zu debuggen, einfache Software wie Bootloader und Betriebssystem-Kernel zu schreiben sowie Reverse Engineering-Software, für die kein Quellcode verfügbar ist. Gelegentlich ist es erforderlich, einige leistungskritische Codeabschnitte manuell zu optimieren. Manchmal wird behauptet, dass ARM-Prozessoren in Assembler nicht programmiert werden können. Daher wird in diesem Tutorial gezeigt, dass dies sehr gut möglich ist, indem gezeigt wird, wie ganze (kleine) Anwendungen vollständig in der ARM-Assemblersprache geschrieben werden!

Da sich die meisten Ressourcen und Tools für ARM auf die C-Programmierung konzentrieren und aufgrund der Komplexität des ARM-Ökosystems die größte Schwierigkeit beim Einstieg in die ARM-Assembly nicht die Sprache selbst ist, sondern die korrekte Verwendung der Tools und das Auffinden der relevanten Dokumentation. Daher konzentriert sich dieses Tutorial auf die Entwicklungsumgebung und darauf, wie der geschriebene Assembler-Code in das endgültige Programm umgewandelt wird. Mit einem guten Verständnis der Umgebung können alle ARM-Anweisungen einfach durch Lesen der Architekturdokumentation erlernt werden.

Aufgrund des komplexen Ökosystems rund um ARM ist eine allgemeine Einführung in den ARM-Prozessormarkt erforderlich.

Über ARM

Arm Holdings ist das Unternehmen hinter der ARM-Architektur. Arm stellt keine Prozessoren selbst her, sondern entwirft die "Blaupausen" für Prozessorkerne, die dann von verschiedenen Halbleiterunternehmen wie ST, TI, NXP und vielen anderen lizenziert werden, die den Prozessor mit verschiedenen Unterstützungshardware kombinieren (insbesondere Flash - und RAM-Speicher) und Peripheriemodule, um einen endgültigen vollständigen Prozessor-IC zu erzeugen. Einige dieser Peripheriemodule sind sogar von anderen Unternehmen lizenziert - zum Beispiel sind die USB-Controller-Module von Synopsys in vielen verschiedenen Prozessoren verschiedener Hersteller zu finden.

Aufgrund dieses Lizenzmodells sind ARM-Prozessorkerne in einer Vielzahl von Produkten enthalten, für die Software mit einem einzigen Satz von Tools (insbesondere Compiler, Assembler und Debugger) entwickelt werden kann. Dies macht Kenntnisse über die ARM-Architektur, insbesondere die ARM-Assemblersprache, für eine Vielzahl von Anwendungen nützlich.

Da für die ARM-Prozessorkerne immer zusätzliche Hardwaremodule erforderlich sind, müssen sowohl der von ARM hergestellte Prozessorkern als auch die herstellerspezifischen Peripheriemodule bei der Entwicklung von Software für ARM-Systeme berücksichtigt werden. Beispielsweise wird der Befehlssatz von ARM definiert und Software-Tools (Compiler, Assembler) müssen für die richtige Befehlssatzversion konfiguriert werden, während die Taktkonfiguration herstellerspezifisch ist und durch einen speziell für einen Prozessor erstellten Initialisierungscode adressiert werden muss .

Architektur und Prozessorvarianten

Die Architektur eines Prozessors definiert die Schnittstelle zwischen Hardware und Software. Sein wichtigster Teil ist der Befehlssatz, aber er definiert auch z.B. Hardware-Verhalten unter außergewöhnlichen Umständen (z. B. Speicherzugriffsfehler, Division durch Null usw.). Prozessorarchitekturen entwickeln sich weiter, sodass sie über mehrere Versionen und Varianten verfügen. Sie definieren auch optionale Funktionen, die in einem Prozessor (z.

B. einer Gleitkommaeinheit) vorhanden sein können oder nicht. Für ARM sind die Architekturen ausführlich in den „ARM Architecture Reference Manuals“ dokumentiert.

Während die Architektur ein abstraktes Konzept ist, ist ein Prozessorkern eine konkrete Definition eines Prozessors (z. B. als Silizium-Layout oder HDL), der eine bestimmte Architektur implementiert. Code, der nur Kenntnisse über die Architektur verwendet (z. B. ein Algorithmus, der auf keine Peripherie zugreift), wird auf jedem Prozessor ausgeführt, der diese Architektur implementiert. Wie bereits erwähnt, entwirft Arm Prozessorkerne für eigene Architekturen. Einige Unternehmen entwickeln jedoch benutzerdefinierte Prozessoren, die einer ARM-Architektur entsprechen, z. B. Apple und Qualcomm.

ARM-Architekturen sind von ARMv1 bis zum neuesten ARMv8 nummeriert. ARMv6 ist die älteste Architektur, die noch in erheblichem Umfang verwendet wird, während ARMv7 die am weitesten verbreitete ist. An die Version werden Suffixe angehängt, um Varianten der Architektur zu kennzeichnen. z.B. ARMv7-M ist für kleine eingebettete Systeme vorgesehen, während ARMv7-A für leistungsstärkere Prozessoren vorgesehen ist. ARMv7E-M bietet Funktionen zur digitalen Signalverarbeitung, einschließlich Sättigungs- und SIMD-Operationen.

Ältere ARM-Prozessoren heißen ARM1, ARM2…, während nach ARM11 der Name „Cortex“ eingeführt wurde. Die Cortex-M-Familie, einschließlich z.B. Cortex-M3 und Cortex-M4 (mit ARMv7-M- bzw. ARMv7E-M-Architektur) wurden für Mikrocontroller entwickelt, bei denen es auf Energieverbrauch, Speichergröße, Chipgröße und Latenz ankommt. Die Cortex-A-Familie, einschließlich z.B. Cortex-A8 und Cortex-A17 (beide mit ARMv7-A-Architektur) sind für leistungsstarke Prozessoren (als "Anwendungsprozessoren" bezeichnet) vorgesehen, z. Multimedia- und Kommunikationsprodukte, insbesondere Smartphones und Tablets. Diese Prozessoren haben viel mehr Rechenleistung, verfügen normalerweise über Schnittstellen mit hoher Bandbreite zur Außenwelt und sind für die Verwendung mit Betriebssystemen auf hoher Ebene, insbesondere Linux (und Android), ausgelegt.

Eine Übersicht über ARM-Prozessoren und ihre implementierte Architekturversion finden Sie unter Wikipedia. Dieses Tutorial konzentriert sich auf die Cortex-M-Mikrocontroller, da diese ohne Betriebssystem viel einfacher zu programmieren sind und die Assemblersprache auf Cortex-A-Prozessoren weniger relevant ist. Die große Auswahl an ARM-basierten Geräten erfordert jedoch Flexibilität bei den Architekturspezifikationen und Softwaretools, was manchmal deren Verwendung erschwert.

Es gibt eigentlich keinen einzigen, sondern drei Befehlssätze für ARM-Prozessoren:

- Der Befehlssatz „A32“ für 32-Bit-ARM-Architekturen, auch einfach als „ARM“ -Befehlssatz bezeichnet, begünstigt die Geschwindigkeit gegenüber dem Verbrauch des Programmspeichers. Alle Anweisungen haben eine Größe von 4 Byte.
- Der Befehlssatz „A64“ gilt für die neuen 64-Bit-ARM-Prozessoren
- Der Befehlssatz „T32“ für 32-Bit-ARM-Architekturen, auch als „Thumb“ bezeichnet, begünstigt den Verbrauch des Programmspeichers gegenüber der Geschwindigkeit. Die meisten Anweisungen haben eine Größe von 2 Byte und einige von 4 Byte.

Die 64-Bit-Cortex-A-Anwendungsprozessoren unterstützen alle drei Befehlssätze, während die 32-Bit-Anwendungsprozessoren nur A32 und T32 unterstützen. Die Cortex-M-Mikrocontroller unterstützen nur T32. In diesem Tutorial wird daher nur auf "thumb2" eingegangen, die zweite Version des Befehlssatzes "T32".

Voraussetzungen

Zunächst müssen geeignete Hard- und Software ausgewählt werden, um die Verwendung der Assemblersprache zu demonstrieren. Für dieses Tutorial ist die Auswahl des spezifischen Mikrocontrollers ohne große Bedeutung. Um jedoch sicherzustellen, dass die Beispielcodes problemlos auf Ihr Setup übertragen werden können, wird empfohlen, dieselben Komponenten zu verwenden.

Auswahl Mikrocontroller

Für den Mikrocontroller ein STM32F103C8 oder ein STM32F103RB von STMicroelectronics verwendet werden. Beide Controller sind bis auf die Flash-Größe (64 KiB vs 128 KiB) und die Anzahl der Pins (48 vs 64) identisch. Diese Controller gehören zur „Mainstream“ -Einsteigerfamilie von ST und sind bei Hobby-Entwicklern mit vielen vorhandenen Online-Ressourcen sehr beliebt. Es sind mehrere Entwicklungsboards mit diesen Controllern verfügbar, zum Beispiel: Nucleo-F103, „/ users / hudakz / code / STM32F103C8T6_Hello / Blue Pill (suchen Sie nach „stm32f103c8t6“ auf AliExpress, Ebay oder Amazon), Quellhardware Olimexino-STM32, STM32-P103, / STM32-H103 / STM32-H103, STM3210E-EVAL.

Prozessortyp & Dokumentation

Zunächst wird anhand der Dokumentation des Mikrocontrollerherstellers ermittelt, welche Art von ARM-Prozessorkern und -architektur für den ausgewählten Chip verwendet wird. Diese Informationen werden verwendet, um alle relevanten Dokumentationen zu finden.

- Die erste Informationsquelle ist das STM32F103RB / C8-Datenblatt. Laut der Überschrift handelt es sich um ein Gerät mittlerer Dichte. Dieser Begriff ist ST-spezifisch und bezeichnet eine Produktfamilie mit bestimmten Merkmalen. Der erste Absatz besagt, dass dieser Mikrocontroller einen Cortex-M3-Prozessorkern mit 72 MHz verwendet. Dieses Dokument enthält auch die elektrischen Eigenschaften und Pinbelegungen.
- Das nächste wichtige Dokument ist das STM32F103 reference manual, das detaillierte Beschreibungen der Peripherie enthält. Insbesondere detaillierte Informationen zu Peripherieregistern und Bits finden Sie hier.
- Die ARM-Entwicklerwebsite enthält Informationen zum Cortex-M3-Prozessorkern, insbesondere zur [1] .arm.com / 100165/0201 / arm_cortexm3_processor_trm_100165_0201_01_de.pdf ARM Cortex-M3-Prozessor - Technisches Referenzhandbuch]. Gemäß Kapitel 1.5.3 implementiert dieser Prozessor die ARMv7-M-Architektur.
- Die Architektur ist im ARMv7M Architecture Reference Manual dokumentiert. Insbesondere enthält es die vollständige Dokumentation des Anweisungssatzes.

Für jede ernsthafte STM32-Entwicklung sollten Sie mit all diesen Dokumenten vertraut sein.

Debug-Adapter

Es gibt viele verschiedene Möglichkeiten, wie Sie Ihr Programm auf einem STM32-Controller ausführen können. Ein Debug-Adapter kann nicht nur Software auf das Flash des Controllers schreiben, sondern auch das Verhalten des Programms analysieren, während es ausgeführt wird. Auf diese Weise können Sie das Programm einzeln ausführen, den Programmfluss und den Speicherinhalt analysieren und die Ursache für Abstürze ermitteln. Die Verwendung eines solchen Debuggers ist zwar nicht unbedingt erforderlich, kann jedoch während der Entwicklung viel Zeit sparen. Da Einstiegsmodelle günstig erhältlich sind, sparen Sie nicht einmal Geld, wenn Sie sie nicht verwenden. Debugger verbinden sich über USB (einige über Ethernet) mit einem Host-PC und über JTAG oder SWD mit dem Mikrocontroller („Ziel“). Während diese beiden Schnittstellen eng miteinander verbunden sind und die gleiche Funktion erfüllen, verwendet SWD weniger Pins (2 statt 4, außer Reset und Masse). Die meisten STM32-Controller unterstützen JTAG und alle unterstützen SWD.

Die Dokumentation aller möglichen Methoden zum Flashen und Debuggen von STM32-Controllern ist nicht Gegenstand dieses Lernprogramms. Viele Informationen zu diesem Thema sind bereits online verfügbar. In diesem Tutorial wird daher davon ausgegangen, dass der Debug-Adapter ST-Link von STMicroelectronics verwendet wird, der billig und bei Hobbyisten beliebt ist . Einige der oben genannten Karten verfügen sogar über einen ST-Link-Adapter, mit dem auch ein extern angeschlossener Mikrocontroller „eigenständig“ flashen kann. Die Beispiele sollten auch mit anderen Adaptern funktionieren. Bitte konsultieren Sie die entsprechende Dokumentation zur Verwendung.

Entwicklungssoftware

Auf der Softwareseite werden mehrere Tools für die Entwicklung der Mikrocontroller-Firmware benötigt. Die Verwendung einer vollständigen integrierten Entwicklungsumgebung (Integrated Development Environment, IDE) spart Zeit und vereinfacht sich wiederholende Schritte, verbirgt jedoch einige wichtige Schritte, die erforderlich sind, um ein grundlegendes Verständnis des Prozesses zu erlangen. Daher wird in diesem Lernprogramm die Verwendung der grundlegenden Befehlszeilentools gezeigt, um die zugrunde liegenden Prinzipien zu demonstrieren. Für eine produktive Entwicklung ist die Verwendung einer IDE natürlich eine vernünftige Wahl. Die vorgestellten Tools funktionieren unter Windows, Linux und Mac OS X (ungetestet).

Zunächst wird ein Texteditor zum Schreiben von Assembly-Code benötigt. Ein guter Editor wie Notepad ++, gedit oder Kate ist ausreichend. Unter Windows kann das Dienstprogramm ST-Link hilfreich sein, ist jedoch nicht unbedingt erforderlich.

Als nächstes wird eine Assembler-Toolchain benötigt, um den geschriebenen Assembler-Code in Maschinencode zu übersetzen. Hierzu wird die GNU Arm Embedded Toolchain verwendet. Dies ist eine Sammlung von Open Source-Tools zum Schreiben von Software in Assembly, C und C ++ für Cortex-M-Mikrocontroller. Obwohl das Paket von ARM verwaltet wird, wird die Software von einer Community von Open-Source-Entwicklern erstellt. Für dieses Tutorial werden nur die enthaltenen Anwendungen "binutils" (einschließlich Assembler & Linker) und "GDB" (Debugger) benötigt. Wenn Sie sich jedoch später für C- oder C ++ - Code entscheiden, sind die enthaltenen Compiler nützlich. Abgesehen davon wird dieses Paket auch als Teil mehrerer IDEs wie SW4STM32, Atollic TrueSTUDIO, emIDE, Embedded Studio und sogar Arduino ausgeliefert. Wenn Sie also (später) mit einem dieser Pakete arbeiten möchten, ist Ihr Assembly-Code kompatibel es.

Eine weitere Komponente ist erforderlich, um mit dem Debug-Adapter zu kommunizieren. Für den ST-Link übernimmt dies OpenOCD, der über USB mit dem Adapter kommuniziert. Andere Adapter wie der J-Link werden mit einer eigenen Software ausgeliefert.

Schließlich kann ein Taschenrechner, der binäre und hexadezimale Modi unterstützt, sehr hilfreich sein. Sowohl der Gnome-Standardrechner als auch der Windows-Rechner (calc.exe) sind geeignet.

Setup

Befolgen Sie die Anweisungen in den nächsten Kapiteln, um Ihre Entwicklungsumgebung einzurichten.

Hardware

Das einzige, was hardwaremäßig erledigt werden muss, ist die Verbindung des Debuggers mit Ihrem Mikrocontroller. Wenn Sie ein Entwicklungsboard mit integriertem Debugger (wie dem Nucleo-F103) verwenden, setzen Sie die Jumper entsprechend (siehe Dokumentation des Boards). Zum Beispiel müssen beim Nucleo-F103 beide „CN2“ -Jumper verbunden sein). Bei Verwendung eines externen Debuggers verbinden Sie die Pins „GND“, „JTMS / SWDIO“ und „JTCK / SWCLK“ des Debuggers und des Mikrocontrollers. Verbinden Sie den "nRESET" -Pin des Debuggers (oder "nTRST", falls er nur diesen hat) mit dem "NRST" -Eingang des Mikrocontrollers.

Wenn Ihr Board über Jumper oder Lötbrücken für den „BOOT0“ -Pin verfügt, stellen Sie sicher, dass der Pin niedrig ist. Die Stromversorgung der Mikrocontroller-Platine erfolgt normalerweise über USB.

Software

Linux

Einige Linux-Distributionen werden mit Paketen für die ARM-Toolchain ausgeliefert. Leider sind diese oft veraltet und auch etwas anders konfiguriert als das von ARM gepflegte Paket. Aus Gründen der Übereinstimmung mit den Beispielen wird dringend empfohlen, das Paket von ARM zu verwenden.

Laden Sie den Linux-Binary-Tarball von der downloads page herunter und extrahieren Sie ihn nach Ein Verzeichnis, dessen Pfad keine Leerzeichen enthält. Das extrahierte Verzeichnis enthält ein Unterverzeichnis namens "bin". Kopieren Sie den vollständigen Pfad in dieses Verzeichnis (z. B. "/ home / user / gcc-arm-none-eabi-8-2019-q3-update / bin").

Fügen Sie diesen Pfad zur Umgebungsvariablen "PATH" hinzu. Auf Ubuntu / Debian-Systemen kann dies erfolgen über:

```
echo 'export PATH = "$ {PATH}: / home / benutzer / gcc-arm-none-eabi-8-2019-q3-update / bin"' | sudo tee /etc/profile.d
/gnu-arm-embedded.sh
```

OpenOCD kann über den Paketmanager installiert werden, z. (Ubuntu / Debian):

```
sudo apt-get installiere openocd
```

Danach melden Sie sich ab und wieder an (oder starten Sie einfach neu). Geben Sie in einem Terminal `arm-none-eabi-as -version` ein. Die Ausgabe sollte ungefähr so aussehen:

```
$ arm-none-eabi-as-version
GNU-Assembler (GNU-Tools für eingebettete Arm-Prozessoren 8-2019-q3-Update) 2.32.0.20190703
Urheberrecht (C) 2019 Free Software Foundation, Inc.
Dieses Programm ist freie Software; Sie können es unter den Bedingungen von weitergeben
die GNU General Public License Version 3 oder höher.
Dieses Programm hat absolut keine Garantie.
Dieser Assembler wurde für ein Ziel von "arm-none-eabi" konfiguriert.
```

Ebenso für `openocd -v` :

```
$ openocd -v
Öffnen Sie den On-Chip-Debugger 0.10.0
Lizenziert unter der GNU GPL v2
Lesen Sie die Fehlerberichte
http://openocd.org/doc/doxygen/bugs.html
```

Wenn eine Fehlermeldung angezeigt wird, ist die Installation nicht korrekt.

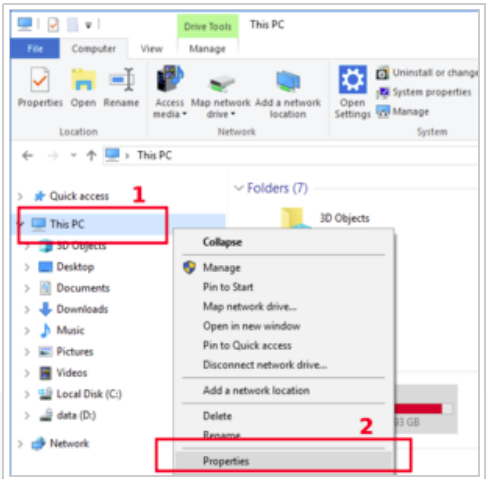
Windows

Laden Sie das Windows-Installationsprogramm von der Website

[2] herunter -tools / gnu-toolchain / gnu-rm / downloads downloads page] und starte es. Aktivieren Sie die Optionen "Pfad zur Umgebungsvariablen hinzufügen" und "Registrierungsinformationen hinzufügen" und deaktivieren Sie "Readme anzeigen" und "gccvar.bat starten".

Ein Windows-Paket für OpenOCD kann von der gnu-mcu-eclipse downloads page heruntergeladen werden. Laden Sie die entsprechende Datei herunter, z. "gnu-mcu-eclipse-openocd-0.10.0-12-20190422-2015-win64.zip". Das Archiv enthält einen Pfad wie "GNU MCU Eclipse / OpenOCD / 0.10.0-12-20190422-2015" Inhalt des inneren Verzeichnisses (dh die Unterverzeichnisse "bin", "doc", "scripts" ...) in ein Verzeichnis, dessen Pfad keine Leerzeichen enthält, zB "C: \ OpenOCD". Sie sollten jetzt ein Verzeichnis "C: \ OpenOCD \ bin "oder ähnliches. Kopieren Sie den vollständigen Pfad.

[Datei: ArmAsmTutorial_SetEnvVar.png | 300px | thumb | right | Umgebungsvariable festlegen]] Legen Sie die Umgebungsvariable "Path" so fest, dass sie diesen Pfad enthält: Klicken Sie mit der rechten Maustaste auf "Dieser PC" und dann auf "Eigenschaften" → "Erweiterte Systemeinstellungen" → "Umgebungsvariablen". Wählen Sie in der unteren Liste (mit der Bezeichnung "Systemvariablen") "Pfad". Klicken Sie auf „Bearbeiten“ → „Neu“, fügen Sie den Pfad ein und klicken Sie mehrmals auf „OK“.



PC-Eigenschaften öffnen

Öffnen Sie ein neues Befehlsfenster (Windows-Taste + R, geben Sie "cmd" + Return ein). Geben Sie `arm-none-eabi-as-version` ein. Die Ausgabe sollte ungefähr so aussehen:

```
C: \> arm-none-eabi-as-version
GNU-Assembler (GNU-Tools für eingebettete Arm-Prozessoren 8-2019-q3-
Update) 2.32.0.20190703
Urheberrecht (C) 2019 Free Software Foundation, Inc.
Dieses Programm ist freie Software; Sie können es unter den
Bedingungen von weitergeben
die GNU General Public License Version 3 oder höher.
Dieses Programm hat absolut keine Garantie.
Dieser Assembler wurde für ein Ziel von "arm-none-eabi" konfiguriert.
```

Ebenso für `openocd -v` :

```
C: \> openocd -v
GNU MCU Eclipse OpenOCD, 64-Bit-Open-On-Chip-Debugger 0.10.0 + dev-00593-g23ad80df4 (2019-04-22-20: 25)
Lizenziert unter der GNU GPL v2
Lesen Sie die Fehlerberichte
    http://openocd.org/doc/doxygen/bugs.html
```

Wenn eine Fehlermeldung angezeigt wird, ist die Installation nicht korrekt.

Schreiben von Baugruppenanwendungen Den vollständigen Quellcode der Beispiele in den folgenden Kapiteln finden Sie unter GitHub. Der Name des entsprechenden Verzeichnisses wird nach jedem folgenden Beispielcode angegeben.

Erstes rudimentäres Programm

Nach dem Software-Setup können Sie mit dem Einrichten eines ersten Projekts beginnen. Erstellen Sie dazu ein leeres Verzeichnis, z. "Prog1".

Erstellen Sie im Projektverzeichnis Ihre erste Assembly-Datei "prog1.S" (".S" ist die Dateinamenerweiterung für Assembly-Dateien im GNU-Kontext) mit folgendem Inhalt:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.word 0x20000400
```

```
.word 0x080000ed
.space 0xe4

nop @ Nichts tun
b. @ Endlosschleife
```

Beispielname: "EmptyProgram"

Wenn diese Datei an den Assembler gesendet wird, werden die Anweisungen in binären Maschinencode mit 2 oder 4 Bytes pro Anweisung übersetzt. Diese Bytes werden zu einem Programmabbild verknüpft, das später in den Flash-Speicher der Steuerung geschrieben wird. Daher beschreibt der Assembler-Code den Inhalt des Flash-Speichers mehr oder weniger direkt.

Die mit einem Punkt "." Beginnenden Zeilen sind Assembler-Anweisungen, die den Assembler-Vorgang steuern. Nur einige dieser Anweisungen geben Bytes aus, die in den Flash-Speicher gelangen. Das @ -Symbol startet einen Kommentar.

In der ersten Zeile kann der Assembler die neue "einheitliche" Befehlssyntax ("UAL" - Unified Assembler Language) anstelle der alten ARM-Syntax verwenden. In der zweiten Zeile wird der verwendete Prozessor Cortex-M3 angegeben, den der Assembler kennen muss, um die auf diesem Prozessor verfügbaren Anweisungen zu erkennen. In der dritten Zeile wird der Assembler angewiesen, den Befehlssatz Thumb (T32) zu verwenden. Wir können nicht sofort damit beginnen, Anweisungen in den Flash-Speicher zu schreiben, da der Prozessor erwartet, dass sich eine bestimmte Datenstruktur ganz am Anfang des Speichers befindet. Dies ist, was die Anweisungen ".word" und ".space" erstellen. Diese werden später erklärt.

Der erste "echte" Befehl ist "nop". Dies ist der erste Befehl, der ausgeführt wird, nachdem der Prozessor gestartet wurde. "Nop" ist die Abkürzung für "No OPeration" (Keine Operation) - der Prozessor unternimmt nichts und fährt mit dem nächsten Befehl fort. Diese nächste Anweisung lautet „b.“. "B" ist die Abkürzung für "Verzweigung" und weist den Prozessor an, zu einem bestimmten "Ziel" -Ort zu springen, d. H. Den Befehl als nächstes an diesem Ziel auszuführen. In der Assemblersprache steht der Punkt „.“ Für die aktuelle Position im Programmspeicher. Daher weist "b" den Prozessor an, zu genau diesem Befehl zu springen, d. H. Ihn immer wieder in einer Endlosschleife auszuführen. Eine solche Endlosschleife findet sich häufig am Ende von Mikrocontroller-Programmen, da sie den Prozessor daran hindert, zufällige Daten auszuführen, die sich nach dem Programm im Flash-Speicher befinden.

Um diesen Assembler-Code zu übersetzen, öffnen Sie ein Terminal- (Linux) / Befehlsfenster (Windows). Rufen Sie das Projektverzeichnis auf, indem Sie `cd <Pfad zum Projektverzeichnis>` eingeben. Rufen Sie den Assembler folgendermaßen auf:

```
arm-none-eabi-as-g prog1.S -o prog1.o
```

Dies weist den Assembler an, die Quelldatei "prog1.S" in eine Objektdatei "prog1.o" zu übersetzen. Dies ist eine Zwischendatei, die binären Maschinencode enthält, jedoch noch kein vollständiges Programm ist. Die Option "-g" weist den Assembler an, Debug-Informationen einzuschließen, die das Programm selbst nicht beeinflussen, aber das Debuggen erleichtern. Rufen Sie den Linker folgendermaßen auf, um diese Objektdatei in ein endgültiges Programm zu verwandeln:

```
arm-none-eabi-ld prog1.o -o prog1.elf -Text = 0x8000000
```

Dadurch wird eine Datei "prog1.elf" erstellt, die das gesamte generierte Programm enthält. Die Option -Ttext weist den Linker an, 0x8000000 als Startadresse des Flash-Speichers zu übernehmen. Der Linker könnte eine Warnung wie diese ausgeben:

```
arm-none-eabi-ld: Warnung: Das Eingabesymbol _start kann nicht gefunden werden. Standardmäßig 0000000008000000
```

Dies ist für die Ausführung des Programms ohne Betriebssystem nicht relevant und kann ignoriert werden.

Programm blinken

Verwenden Sie OpenOCD wie folgt, um die kompilierte Anwendung auf den über ST-Link angeschlossenen Mikrocontroller herunterzuladen:

```
openocd -f Schnittstelle / stlink-v2.cfg -f Ziel / stm32f1x.cfg -c "Programm prog1.elf Verifiziere Reset-Exit"
```

Leider macht die Anwendung nichts, was von außen beobachtet werden kann, außer vielleicht den Stromverbrauch zu erhöhen.

Starten des Debuggers

Um zu überprüfen, ob das Programm tatsächlich ausgeführt wird, starten Sie eine Debugsitzung, um das Verhalten des Prozessors genau zu beobachten. Führen Sie OpenOCD zunächst so aus, dass es als GDB-Server fungiert:

```
openocd -f Schnittstelle / stlink-v2.cfg -f Ziel / stm32f1x.cfg
```

Öffnen Sie dann ein neues Terminal- / Befehlsfenster und starten Sie eine GDB-Sitzung:

```
arm-none-eabi-gdb prog1.elf
```

GDB bietet eine eigene interaktive textbasierte Benutzeroberfläche. Geben Sie zunächst diesen Befehl ein, damit sich GDB mit der bereits ausgeführten OpenOCD-Instanz verbindet:

```
Zielfernbedienung: 3333
```

Beenden Sie dann das aktuell ausgeführte Programm:

```
Monitor zurücksetzen halt
```

Wenn dies fehlschlägt, halten Sie die Reset-Taste Ihres Boards kurz vor der Ausführung des Befehls gedrückt und wiederholen Sie den Vorgang, bis er erfolgreich ist. GDB kann auch Code in den Flash-Speicher herunterladen, indem Sie einfach Folgendes eingeben:

```
Belastung
```

Welches überschreibt das zuvor geflashte Programm (das in diesem Fall sowieso identisch ist). Setzen Sie die Steuerung nach dem Laden des Programms erneut zurück:

```
Monitor zurücksetzen halt
```

Untersuchen Sie nun den Inhalt der CPU-Register:

```
info reg
```

Die Ausgabe sollte ungefähr so aussehen

```
r0 0x0 0
r1 0x0 0
r2 0x0 0
r3 0x0 0
r4 0x0 0
r5 0x0 0
r6 0x0 0
r7 0x0 0
r8 0x0 0
r9 0x0 0
r10 0x0 0
r11 0x0 0
r12 0x0 0
sp 0x0 0x0
lr 0x0 0
pc 0x8000000 0x8000000 <_stack + 133693440>
xPSR 0x1000000 16777216
msp 0x20000400 0x20000400
psp 0x27e3fa34 0x27e3fa34
Primask 0x0 0
basepri 0x0 0
Fehlermaske 0x0 0
steuern Sie 0x0 0
```

Zu diesem Zeitpunkt ist der Prozessor bereit, die Ausführung Ihres Programms zu starten. Der Prozessor wird

kurz vor dem ersten Befehl angehalten, der "nop" ist. Sie können den Prozessor eine einzelne Anweisung (d. H. Das "nop") ausführen lassen, indem Sie Folgendes eingeben

```
stepi
```

Wenn Sie erneut `info reg` eingeben, sehen Sie, dass der PC jetzt "0x80000ee" ist, d. H. Der Prozessor ist im Begriff, den nächsten Befehl "b." Auszuführen. Wenn Sie

```
stepi
```

erneut (wiederholt) ausführen, passiert nichts mehr - der Controller steckt genau wie vorgesehen in der erwähnten Endlosschleife. Sie können den Prozessor anweisen, das Programm kontinuierlich auszuführen, ohne nach jeder Anweisung durch Eingabe anzuhalten

```
fortsetzen
```

Sie können das laufende Programm unterbrechen, indem Sie „Strg + C“ drücken. Führen Sie die Befehle aus

```
töten
Verlassen
```

um GDB zu verlassen. Sie können OpenOCD beenden, indem Sie im Terminal die Tastenkombination "Strg + C" drücken.

Verwenden von Prozessorregistern

Das Beispielprogramm hat nichts Nützliches getan, aber jedes "echte" Programm muss einige Daten verarbeiten. Bei ARM erfolgt die Datenverarbeitung über die Prozessorregister. Die 32-Bit-ARM-Plattformen verfügen über 16 Prozessorregister mit einer Größe von jeweils 32 Bit. Die letzten drei davon (r13-r15) haben eine besondere Bedeutung und können nur mit bestimmten Einschränkungen verwendet werden. Die ersten 13 (r0-r12) können vom Anwendungscode für die Datenverarbeitung frei verwendet werden.

Alle Berechnungen (z. B. Addition, Multiplikation, Logik und / oder) müssen an diesen Prozessorregistern durchgeführt werden. Um Daten aus dem Speicher zu verarbeiten, müssen sie zuerst in ein Register geladen, dann verarbeitet und wieder im Speicher abgelegt werden. Dies ist typisch für RISC-Plattformen und wird als "Load-Store-Architektur" bezeichnet.

Als Ausgangspunkt für jede Berechnung müssen einige spezifische Werte in die Register eingetragen werden. Der einfachste Weg dies zu tun ist:

```
ldr r0 = 123456789
```

Die Nummer 123456789 wird als Teil des Programms codiert, und der Befehl lässt den Prozessor sie in das Register "r0" kopieren. Stattdessen kann eine beliebige Zahl und ein beliebiges Register im Bereich von r0 bis r13 verwendet werden.

Mit der Anweisung „mov“ kann der Inhalt von einem Register in ein anderes kopiert werden:

```
mov r1, r0
```

Dies kopiert r0 nach r1. Im Gegensatz zu einigen anderen Prozessorarchitekturen kann mit "mov" nicht auf den Speicher zugegriffen werden, sondern nur auf die Prozessorregister.

In ARM werden 32-Bit-Zahlen als "Wörter" bezeichnet und am häufigsten verwendet. 16-Bit-Zahlen werden wie üblich als Halbwörter und 8-Bit-Zahlen als Bytes bezeichnet.

Zugriff auf Peripherie

Zum Schreiben von Mikrocontroller-Programmen, die mit der Außenwelt interagieren, ist der Zugriff auf die Peripheriemodule des Controllers erforderlich. Die Interaktion mit der Peripherie erfolgt hauptsächlich über Peripherieregister (auch als SFR (Special Function Register) bezeichnet). Trotz ihres Namens arbeiten sie ganz anders als Prozessorregister. Anstelle von Zahlen haben sie Adressen (im Bereich von 0x40000000-0x50000000), die nicht zusammenhängend sind (d. H. Es gibt Lücken). Sie können nicht direkt für die Datenverarbeitung verwendet werden, sondern müssen vor und nach Berechnungen explizit gelesen und

geschrieben werden. Nicht alle von ihnen sind 32-Bit; Viele haben nur 16 Bit, und einige dieser Bits sind möglicherweise nicht vorhanden und es kann nicht auf sie zugegriffen werden. In der Dokumentation des Mikrocontrollerherstellers werden Namen für diese Register verwendet, die der Assembler jedoch nicht kennt. Daher muss der Assemblycode die numerischen Adressen verwenden.

Die einfachste Möglichkeit, den Mikrocontroller dazu zu bringen, etwas zu tun, das ein sichtbares Ergebnis liefert, besteht darin, ein Signal über einen Ausgangspin zu senden, um eine LED einzuschalten. Die Verwendung eines Pins zum Senden / Empfangen von beliebigen, durch Software definierten Signalen wird als „GPIO“ (General Purpose Input / Output) bezeichnet. Wählen Sie zunächst einen Pin aus, z. B. PA8 (dieser ist bei allen Gehäusevarianten verfügbar). Schließen Sie eine LED an diesen Pin und an GND ("active high") an. Verwenden Sie einen Vorwiderstand, um den Strom auf max. 15 mA (das absolute Maximum ist 25 mA), z.B. 100Ω für eine 3,3V Versorgung und eine Standard LED. Verwenden Sie für höhere Lasten (z. B. Hochleistungs-LEDs oder ein Relais) einen geeigneten Transistor.

Wie bei den meisten Mikrocontrollern sind die Pins in sogenannten "Ports" zusammengefasst, von denen jeder bis zu 16 Pins hat. Die Ports werden durch Buchstaben des Alphabets benannt, d. H. "GPIOA", "GPIOB", "GPIOC" usw. Die Anzahl der Ports und Pins variiert zwischen den einzelnen Mikrocontrollertypen. Die 16 Pins eines Ports können in einem Schritt gelesen oder beschrieben werden.

Uhrenkonfiguration

Viele ARM-Controller verfügen über eine bestimmte Falle: Die meisten Peripheriemodule sind standardmäßig deaktiviert, um Energie zu sparen. Die Software muss die benötigten Module explizit freischalten. Bei STM32-Steuerungen erfolgt dies über das Modul „RCC“ (Reset and Clock Control). Insbesondere ermöglicht dieses Modul der Software, das Taktsignal für jedes Peripheriemodul zu deaktivieren / aktivieren. Da MOSFET-basierte Schaltungen (praktisch alle modernen ICs) nur dann Strom ziehen, wenn ein Taktsignal angelegt wird, kann das Ausschalten des Takts nicht verwendeter Module den Stromverbrauch erheblich reduzieren.

Der Zugriff und die Konfiguration der GPIO-Pins erfolgt über die Register der GPIO-Peripherie. Das STM32 verfügt über mehrere identische Instanzen von GPIO-Modulen, die als GPIOA, GPIOB usw. bezeichnet werden. Jede dieser Instanzen verfügt über eine eigene Basisadresse, die in Kapitel 3.3 des Referenzhandbuchs erneut beschrieben wird (z. B. „0x40010800“ für GPIOA, „0x40010C00“). für GPIOB usw.). Die Register des GPIO-Moduls sind in Kapitel 9.2 beschrieben, und es gibt eine Instanz jedes Registers pro GPIO-Modul. Um auf ein bestimmtes Register eines bestimmten GPIO-Moduls zuzugreifen, muss die Basisadresse dieses Moduls zur Offset-Adresse des Registers hinzugefügt werden. Beispielsweise hat "GPIOA_IDR" die Adresse "0x40010800 + 0x08 = 0x40010808", während "GPIOB_ODR" die Adresse "0x40010C00 + 0x0C = 0x40010C0C" hat.

Die Konfiguration der einzelnen GPIO-Pins erfolgt über die Register „GPIOx_CRL“ und „GPIOx_CRH“ („x“ ist ein Platzhalter für das konkrete GPIO-Modul) - siehe Kapitel 9.2.1 und 9.2.2. Beide Register sind identisch aufgebaut, wobei jeder Pin 4 Bits verwendet, sodass jedes der beiden Register 8 Pins in 8x4 = 32 Bits verarbeitet. Die Pins 0-7 werden von "GPIOx_CRL" und die Pins 8-15 von "GPIOx_CRH" konfiguriert. Pin 0 wird durch die Bits 0-3 von „GPIOx_CRL“, Pin 1 durch die Bits 4-7 von „GPIOx_CRL“, Pin 8 durch die Bits 0-3 von „GPIOx_CRH“ usw. konfiguriert.

Die 4 Bits pro Pin sind in zwei 2-Bit-Felder aufgeteilt: "MODE" belegt die Bits 0-1 und "CNF" die Bits 2-3. “MODE” wählt zwischen Eingangs- und Ausgangsmodus (mit unterschiedlichen Geschwindigkeiten). Im Ausgangsmodus bestimmt „CNF“, ob der Ausgangswert über die Software konfiguriert wird (Modus „General Purpose“) oder von einem anderen Peripheriemodul gesteuert wird (Modus „Alternate Function“) und ob zwei Transistoren (Modus „Push-Pull“) oder Eine ("Open-Drain") wird verwendet, um den Ausgang anzusteuern. Im Eingangsmodus wählt „CNF“ zwischen Analogmodus (für ADC), potentialfreiem Eingang und Eingang mit Pull-Up / Down-Widerständen (abhängig vom Wert im Register „GPIOx_ODR“).

Daher müssen zum Konfigurieren von Pin PA8 in den Modus "Push-Pull für allgemeine Ausgabe, 2 MHz" die Bits 0-3 von "GPIOA_CRH" auf den Wert "2" gesetzt werden. Der Standardwert von "4" konfiguriert den Pin als "Eingang". Um die anderen Pins auf ihrer "Input" -Konfiguration zu halten, muss der Wert "0x44444442" in das Register "GPIOA_CRH" mit der Adresse "0x40010804" geschrieben werden:

```
ldr r0, = 0x44444442
ldr r1 = 0x40010804
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2
```

GPIO-Pins schreiben

Der GPIO-Pin gibt weiterhin den Standardwert aus, der für "niedrig" 0 ist. Um die LED einzuschalten, muss der Ausgang für „high“ auf „1“ gesetzt werden. Dies wird über das GPIOA_ODR-Register erreicht, das 16 Bit hat, eines für jeden Pin (siehe Kapitel 9.2.4). Um die LED zu aktivieren, setzen Sie Bit 8 auf eins:

```
.syntax vereinheitlicht
```

```
.cpu cortex-m3
.Daumen

.word 0x20000400
.word 0x080000ed
.space 0xe4

ldr r0, = 0x00000004
ldr r1 = 0x40021018
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren

ldr r0, = 0x44444442
ldr r1 = 0x40010804
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2

ldr r0, = 0x100
ldr r1 = 0x4001080C
str r0, [r1] @ Setzen Sie ODR8 in GPIOA_ODR auf 1, um PA8 hoch zu setzen

b.
```

Beispielname: "SetPin"

Dieses Programm aktiviert den GPIOA-Peripherietakt, konfiguriert PA8 als Ausgang und setzt ihn auf hoch. Wenn Sie es auf Ihrem Mikrocontroller ausführen, sollte die LED aufleuchten - das erste Programm, das einen sichtbaren Effekt hat!

Datenverarbeitung

ARM unterstützt viele Anweisungen für mathematische Operationen. Beispielsweise kann das Hinzufügen wie folgt durchgeführt werden:

```
ldr r0 = 222
ldr r1 = 111
addiere r2, r0, r1
```

Dies lädt zuerst den Wert 222 in das Register r0, lädt 111 in r1 und addiert schließlich r0 und r1 und speichert das Ergebnis (d. H. 333) in r2. Der Operand für das Ergebnis steht (fast) immer links, während der oder die Eingabeoperanden rechts folgen.

Sie können auch ein Eingaberegister mit dem Ergebnis überschreiben:

```
addiere r0, r0, r1
```

Dadurch wird das Ergebnis in r0 geschrieben und der vorherige Wert überschrieben. Dies wird üblicherweise auf abgekürzt

```
addiere r0, r1
```

Der Ausgabeoperand kann weggelassen werden und die erste Eingabe (hier: r0) wird überschrieben. Dies gilt für die meisten Datenverarbeitungsanweisungen. Andere häufig verwendete Datenverarbeitungsanweisungen, die auf ähnliche Weise verwendet werden, sind:

- '*sub*' für die Subtraktion
- '*mul*' zur Multiplikation
- '*und*' für bitweise und
- '*orr*' für bitweises oder
- '*eor*' für bitweises exklusives oder ("xor")
- '*lsl*' für logische Linksverschiebung
- '*lsr*' für logische Rechtsverschiebung

Die meisten dieser Anweisungen können nicht nur Register als Eingabe, sondern auch unmittelbare Argumente verwenden. Ein solches Argument wird direkt in den Befehl codiert, ohne dass es zuerst in ein Register geschrieben werden muss. Direkten Argumenten muss ein Hash-Zeichen # vorangestellt werden. Sie können dezimal, hexadezimal oder binär sein. Zum Beispiel,

```
addiere r0, r0, # 23
```

addiert 23 zum Register r0 und speichert das Ergebnis in r0. Dies kann wieder auf gekürzt werden

```
addiere r0, # 23
```

Solche unmittelbaren Argumente können nicht beliebig groß sein, da sie in den Befehl passen müssen, der 16 oder 32 Bit groß ist und auch etwas Platz für den Befehl und die Registernummern benötigt. Wenn Sie also eine große Zahl hinzufügen möchten, müssen Sie zuerst "ldr" wie gezeigt verwenden, um sie in ein Register zu laden.

Probieren Sie die obigen Beispiele aus und verwenden Sie GDB, um deren Verhalten zu untersuchen. Verwenden Sie den Befehl "info reg" der GDB, um den Inhalt des Registers anzuzeigen. Vergessen Sie nicht, die Befehle "arm-none-eabi-as" und "arm-none-eabi-ld" auszuführen, um das Programm zu übersetzen.

Peripherieregister lesen

Das letzte Beispiel funktioniert, hat aber einen Fehler: Obwohl nur wenige Bits pro Register geändert werden müssen, überschreibt der Code alle Bits im Register auf einmal. Die Bits, die nicht geändert werden sollen, werden nur mit ihrem jeweiligen Standardwert überschrieben. Wenn einige dieser Bits zuvor geändert wurden, um beispielsweise ein anderes Peripheriemodul zu aktivieren, gehen diese Änderungen verloren. Den Status des Registers während des gesamten Programms zu verfolgen, ist kaum praktikabel. Da ARM das Modifizieren einzelner Bits nicht zulässt, besteht die Lösung darin, das gesamte Register zu lesen, die Bits nach Bedarf zu modifizieren und das Ergebnis zurückzuschreiben. Dies wird als "Lesen-Ändern-Schreiben" -Zyklus bezeichnet.

Das Lesen der Register erfolgt über die Anweisung "ldr". Wie bei "str" muss die Adresse zuvor in ein Prozessorregister geschrieben werden, und der Befehl speichert die gelesenen Daten auch in einem Prozessorregister. Ab dem Register „RCC_APB2ENR“ können Sie es lesen über:

```
ldr r1 = 0x40021018
ldr r0, [r1]
```

Obwohl die beiden "ldr" -Anweisungen ähnlich aussehen, funktionieren sie unterschiedlich - die erste lädt einen festen Wert in ein Register (r1), während die zweite Daten aus dem Peripherieregister in r1 lädt.

Der geladene Wert sollte dann durch Setzen von Bit 2 auf "1" geändert werden. Dies kann mit der Anweisung "orr" erfolgen:

```
orr r0, r0, # 4
```

Danach können wir r0 wie bisher speichern.

Das GPIOA_CRH-Register ist etwas komplizierter: Die Bits 0, 2 und 3 müssen gelöscht werden, während Bit 1 auf 1 gesetzt werden muss. Die anderen Bits (4-31) müssen ihren Wert beibehalten. Um die Bits zu löschen, verwenden Sie nach dem Laden des aktuellen Peripherieregisterwerts die Anweisung „und“:

```
ldr r1 = 0x40010804
ldr r0, [r1]
und r0, # 0xfffffff0
oder r0, # 2
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2
```

Für die „GPIOx_ODR“ -Register sind solche Tricks nicht erforderlich, da es ein spezielles „GPIOx_BSRR“ -Register gibt, das das Schreiben einzelner Bits vereinfacht: Dieses Register kann nicht gelesen werden, und das Schreiben von Nullen in ein beliebiges Bit hat keine Auswirkungen auf den GPIO-Status. Wenn jedoch eine 1 in eines der Bits 0-15 geschrieben wird, wird der entsprechende GPIO-Pin auf hoch gesetzt (d. H. Das entsprechende Bit in ODR wird auf 1 gesetzt). Wenn eines der Bits 16-31 auf 1 geschrieben wird, wird der entsprechende Pin auf niedrig gesetzt. Der Pin kann also wie folgt auf 1 gesetzt werden:

```
ldr r1 = 0x40010810
ldr r0, = 0x100
str r0, [r1] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen
```

Das geänderte Programm lautet also:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.word 0x20000400
.word 0x080000ed
.space 0xe4

ldr r1 = 0x40021018
ldr r0, [r1]
orr r0, r0, # 4
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren

ldr r1 = 0x40010804
ldr r0, [r1]
und r0, # 0xfffffff0
oder r0, # 2
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2

ldr r1 = 0x40010810
ldr r0, = 0x100
str r0, [r1] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

b.
```

Beispielname: "SetPin2"

Sprunganweisungen

Für ein traditionelles "Hallo Welt" -Erlebnis sollte die LED nicht nur aufleuchten, sondern auch blinken, d. H. Wiederholt ein- und ausschalten. Das Setzen des Pins PA8 auf einen niedrigen Pegel kann erreicht werden, indem eine 1 bis Bit 24 in das Register "GPIO_BSRR" geschrieben wird:

```
ldr r1 = 0x40010810
ldr r0, = 0x1000000
str r0, [r1]
```

Wenn Sie dies hinter die Anweisungen zum Einschalten der LED einfügen, wird sie ein- und wieder ausgeschaltet. Damit die LED blinkt, müssen diese beiden Blöcke endlos wiederholt werden, d. H. Am Ende des Codes muss eine Anweisung zum Zurückspringen zum Anfang vorhanden sein.

Eine einfache Endlosschleife wurde bereits erklärt: Die Anweisung „b.“, Die sich nur wiederholt ausführt. Damit es an eine andere Stelle springt, muss der Punkt durch die gewünschte Zieladresse ersetzt werden. Beispiel:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.word 0x20000400
.word 0x080000ed
.space 0xe4

ldr r1 = 0x40021018
ldr r0, [r1]
orr r0, r0, # 4
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren

ldr r1 = 0x40010804
ldr r0, [r1]
und r0, # 0xfffffff0
oder r0, # 2
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2

ldr r1 = 0x40010810
ldr r0, = 0x100
str r0, [r1] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

ldr r1 = 0x40010810
ldr r0, = 0x1000000
str r0, [r1] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen
```

```
b 0x8000104
```

Beispielname: "Blink"

Die angegebene Adresse ist eine absolute Adresse. Dies ist die Adresse des Befehls "ldr" am Anfang des Blocks, um den Pin auf "high" zu setzen. Tatsächlich kann der Verzweigungsbefehl "b" nicht direkt zu einer solchen absoluten Adresse springen, da eine 32 Bit breite Adresse nicht in einem 16/32 Bit breiten Befehl codiert werden kann. Stattdessen berechnet der Assembler die Entfernung des Sprungziels und den Ort des Befehls "b" und speichert ihn im Befehl. Beim Rückwärtsspringen ist dieser Abstand negativ.

Bei der Ausführung von Programmcode speichert der Prozessor immer die Adresse des aktuell ausgeführten Befehls plus vier im Register r15, das daher auch als PC bezeichnet wird, dem Programmzähler. Bei einem "b" -Befehl addiert der Prozessor den enthaltenen Abstandswert zum PC-Wert, um die absolute Adresse des Sprungziels zu berechnen, bevor er dorthin springt.

Dies bedeutet, dass "b" einen relativen Sprung ausführt, und selbst wenn der gesamte Maschinencodeabschnitt an eine andere Stelle im Speicher verschoben würde, würde der Code weiterhin funktionieren. Die Assemblersyntax stellt dies jedoch nicht wirklich dar, da der Assembler absolute Adressen erwartet, die er dann in relative Adressen umwandelt.

Die Angabe der Zieladresse direkt wie gezeigt ist sehr unpraktisch, da sie manuell berechnet werden muss. Wenn der Codeabschnitt verschoben oder geändert wird, muss die Adresse geändert werden. Um dies zu korrigieren, unterstützt der Assembler Labels: Sie können einem bestimmten Codestandort einen Namen zuweisen und diesen Namen verwenden, um auf den Codestandort zu verweisen, anstatt die Adresse als Zahl anzugeben. Ein Label wird definiert, indem sein Name gefolgt von einem Doppelpunkt geschrieben wird:

```
BlinkLoop:

ldr r1 = 0x40010810
ldr r0, = 0x100
str r0, [r1] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

ldr r1 = 0x40010810
ldr r0, = 0x1000000
str r0, [r1] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

b BlinkLoop
```

Beispielname: „Blink2“

Dies ist eine reine Funktion des Assemblers - der generierte Maschinencode ist identisch mit dem vorherigen Beispiel. In „b BlinkLoop“ ersetzt der Assembler die Bezeichnung für die Adresse, die er darstellt, um die relative Sprungdistanz zu berechnen. Der Assembler bietet tatsächlich keine direkte Möglichkeit, den relativen Versatz, der in der Anweisung codiert wird, direkt anzugeben. Dies kann jedoch folgendermaßen erfolgen:

```
b (. + 4 + 42 * 2)
```

Die resultierende Anweisung enthält "42" als Sprungoffset. Wie in der Syntax angegeben, multipliziert der Prozessor diese Zahl mit 2 (da Befehle nur an geraden Speicheradressen gespeichert werden können, würde ein Speicherbit für die direkte Angabe der Zahl verschwendet) und fügt die Adresse des Befehls "b" plus hinzu 4. Die Assemblersyntax soll das Endergebnis der Operation darstellen, sodass der Assembler die eigentümlichen Vorberechnungen des Prozessors rückgängig macht. Wenn Sie diese Berechnung selbst durchführen möchten, müssen Sie die eigene Berechnung des Assemblers mit dem oben gezeigten Ausdruck erneut rückgängig machen. Es gibt jedoch normalerweise keinen Grund, dies zu tun.

Zählschleifen

Das obige Beispiel für eine blinkende LED funktioniert noch nicht wirklich - die LED blinkt so schnell, dass das menschliche Auge sie nicht sehen kann. Die LED leuchtet nur schwach. Um eine korrekte Blinkfrequenz zu erreichen, muss der Code verlangsamt werden. Am einfachsten ist es, wenn der Prozessor zwischen dem Setzen des Pins auf High und Low eine große Anzahl von "Dummy" -Anweisungen ausführt. Das einfache Platzieren vieler "NOP" -Anweisungen ist jedoch nicht möglich, da einfach nicht genügend Programmspeicher vorhanden ist, um alle zu speichern. Die Lösung ist eine Schleife, die dieselben Anweisungen eine bestimmte Anzahl von Malen ausführt (im Gegensatz zu den Endlosschleifen aus den obigen Beispielen). Dazu muss der Prozessor die Anzahl der Schleifeniterationen zählen. Tatsächlich ist es einfacher, "abwärts" als "aufwärts" zu zählen. Laden Sie also zunächst die gewünschte Anzahl von Iterationen in ein Register und beginnen Sie die Schleife, indem Sie "1" abziehen:

```
ldr r2 = 1000000
subs r2, # 1
```

Jetzt sollte der Prozessor eine Entscheidung treffen: Wenn das Register Null erreicht hat, beenden Sie die Schleife; ansonsten fahren Sie fort, indem Sie erneut „1“ abziehen. Die ARM-Mathematikbefehle können automatisch einige Tests für das Ergebnis durchführen, um zu überprüfen, ob es positiv / negativ oder null ist und ob ein Überlauf aufgetreten ist. Um diese Überprüfungen zu aktivieren, fügen Sie dem Befehlsnamen ein "s" hinzu - daher "subs" anstelle von "sub". Das Ergebnis dieser Überprüfungen wird automatisch im „Application Program Status Register“ (APSR) gespeichert - die enthaltenen Bits N, Z, C, V zeigen an, ob das Ergebnis negativ war, Null, das Übertragsbit gesetzt oder einen Überlauf verursacht hat. Auf dieses Register wird normalerweise nicht direkt zugegriffen. Verwenden Sie stattdessen die bedingte Variante der Anweisung "b", bei der zwei Buchstaben angehängt werden, um die gewünschte Bedingung anzugeben. Der Sprung wird nur ausgeführt, wenn die Bedingung erfüllt ist. Andernfalls führt der Befehl nichts aus. Die verfügbaren Bedingungscodes werden im Kapitel „Bedingungscodes“ dieses Tutorials beschrieben. Die Bedingungen sind in Bezug auf die erwähnten Bits des APSR formuliert. Beispielsweise führt der Befehl "bne" nur dann einen Sprung aus, wenn das Flag "Z" nicht gesetzt ist, dh wenn das Ergebnis des letzten Mathematikbefehls (mit einem angehängten "s") "nicht" war. Null. Der "beq" -Befehl ist das Gegenteil davon - er führt nur dann einen Sprung aus, wenn das Ergebnis "" Null war.

Wenn Sie also zum Anfang der Schleife zurückspringen möchten, fügen Sie vor dem Befehl "subs" ein Label hinzu und setzen Sie nach dem Befehl "subs", der zu diesem Label springt, einen Befehl "bne", wenn der Zähler noch nicht Null erreicht hat:

```
ldr r2 = 1000000
delay1:
subs r2, # 1
bne delay1 @ Verzögerungsschleife durchlaufen
```

Die eigentliche Schleife besteht nur aus den beiden Befehlen "subs" und "bne". Durch Platzieren von zwei dieser Schleifen (mit zwei verschiedenen Beschriftungen!) Zwischen den Blöcken, mit denen die Stifte ein- und ausgeschaltet werden, wird die Blinkfrequenz so weit verringert, dass sie sichtbar wird:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.word 0x20000400
.word 0x080000ed
.space 0xe4

ldr r1 = 0x40021018
ldr r0, [r1]
orr r0, r0, # 4
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren

ldr r1 = 0x40010804
ldr r0, [r1]
und r0, # 0xfffffff0
oder r0, # 2
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2

BlinkLoop:
ldr r1 = 0x40010810
ldr r0, = 0x100
str r0, [r1] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

ldr r2 = 1000000
delay1:
subs r2, # 1
bne delay1 @ Verzögerungsschleife durchlaufen

ldr r1 = 0x40010810
ldr r0, = 0x1000000
str r0, [r1] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

ldr r2 = 1000000
delay2:
subs r2, # 1
bne delay2 @ Verzögerungsschleife durchlaufen
```

```
b BlinkLoop
```

Beispielname: "BlinkDelay"

Möglicherweise stellen Sie fest, dass die Register r0-r2 immer wieder mit denselben Werten geladen werden. Um den Code sowohl kürzer als auch schneller zu machen, nutzen Sie die verfügbaren Prozessorregister und laden Sie die Werte, die sich nicht ändern, vor der Schleife. Dann benutze sie einfach in der Schleife:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.word 0x20000400
.word 0x080000ed
.space 0xe4

ldr r1 = 0x40021018
ldr r0, [r1]
orr r0, r0, # 4
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren

ldr r1 = 0x40010804
ldr r0, [r1]
und r0, # 0xfffffff0
oder r0, # 2
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2


ldr r0, = 0x40010810 @ Adresse von GPIOA_BSRR laden
ldr r1, = 0x100 @ Wert registrieren, um Pin auf High zu setzen
ldr r2, = 0x1000000 @ Wert registrieren, um Pin auf Low zu setzen
ldr r3, = 1000000 @ Iterationen für die Verzögerungsschleife

BlinkLoop:
str r1, [r0] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

mov r4, r3
delay1:
subs r4, # 1
bne delay1 @ Verzögerungsschleife durchlaufen

str r2, [r0] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

mov r4, r3
delay2:
subs r4, # 1
bne delay2 @ Verzögerungsschleife durchlaufen

b BlinkLoop
```

Beispielname: "BlinkDelay2"

Verwenden von RAM

Bisher wurden alle Daten in den Beispielcodes in Peripherie- oder Prozessorregistern gespeichert. Mit Ausnahme der einfachsten Programme müssen größere Datenmengen verarbeitet werden, für die die dreizehn Allzweck-Prozessorregister nicht ausreichen. Zu diesem Zweck verfügt der Mikrocontroller über einen SRAM-Block, der 20 KB Daten speichert. Der Zugriff auf Daten im RAM funktioniert ähnlich wie der Zugriff auf Peripherieregister. Laden Sie die Adresse in ein Prozessorregister und verwenden Sie "ldr" und "str", um die Daten zu lesen und zu schreiben. Nach dem Zurücksetzen enthält der RAM nur zufällige Einsen und Nullen, so dass vor dem ersten Lesezugriff ein bestimmter Wert gespeichert werden muss.

Wenn der Programmierer entscheidet, welche Daten wo abgelegt werden sollen, muss er verfolgen, welche Adresse in welchem Speicher welche Daten enthält. Mit dem Assembler können Sie den Überblick behalten, indem Sie angeben, welche Art von Speicherblöcken Sie benötigen, und ihnen Namen geben. Dazu müssen Sie dem Assembler zunächst mitteilen, dass sich die nächsten Anweisungen auf Daten beziehen und nicht auf Anweisungen mit der Anweisung „.data“. Verwenden Sie dann die Anweisung ".space" für jeden benötigten Speicherblock. Um den Blöcken Namen zuzuweisen, setzen Sie eine Beschriftungsdefinition (mit einem Doppelpunkt) vor diese. Fügen Sie nach den Definitionen eine ".text" -Anweisung ein, um sicherzustellen, dass die Anweisungen danach ordnungsgemäß in den Programmspeicher (Flash) übertragen werden:

```
.Daten
var1:
.space 4 @ Reserve 4 Bytes für den Speicherblock "var1"
var2:
.space 1 @ Reserve 1 Byte für Speicherblock "var2"

.Text
@ Anleitung geh hier ...
```

Hier ist ein Datenblock von 4 Bytes reserviert und heißt "var1". Ein weiterer Block von 1 Byte heißt "var2". Beachten Sie, dass durch das Einfügen dieser Zeilen die Assembler-Ausgabe nicht geändert wird. Dies sind lediglich Anweisungen für den Assembler. Um auf diese Speicherblöcke zuzugreifen, können Sie "var1" und "var2" wie wörtliche Adressen verwenden. Lade sie in die Register und benutze diese mit "ldr" und "str" wie folgt:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.word 0x20000400
.word 0x080000ed
.space 0xe4

.Daten
var1:
.space 4 @ Reserve 4 Bytes für den Speicherblock "var1"
var2:
.space 1 @ Reserve 1 Byte für Speicherblock "var2"

.Text


ldr r0, = var1 @ Adresse von var1 abrufen
ldr r1, = 0x12345678
str r1, [r0] @ Speichere 0x12345678 in Speicherblock "var1"


ldr r1, [r0] @ Speicherblock "var1" lesen
und r1, # 0xFF @ Setze die Bits 8..31 auf Null
ldr r0, = var2 @ Adresse von var2 abrufen
strb r1, [r0] @ Speichere ein einzelnes Byte in var2

b.
```

Beispielname: "RAMVariables"

Beachten Sie die Verwendung von "strb" - es funktioniert ähnlich wie "str", speichert jedoch nur ein einzelnes Byte. Da das Prozessorregister r1 natürlich 32 Bit groß ist, werden nur die unteren 8 Bit gespeichert, und der Rest wird ignoriert.

Es fehlt noch etwas - nirgendwo im Code befindet sich eine Adresse des RAM. Um dem Linker mitzuteilen, wo sich der RAM befindet, übergeben Sie die Option `-Tdata = 0x20000000` an den Aufruf `arm-none-eabi-ld` , um dem Linker mitzuteilen, dass dies die Adresse ist des ersten Bytes des RAM.

Wenn Sie dieses Programm über GDB ausführen, können Sie die im Speicher abgelegten Daten mit den Befehlen `x / 1xw & var1` und `x / 1xb & var2` lesen. Nach dieser kurzen Einführung wird eine abstraktere Übersicht angezeigt.

Speicherverwaltung

Wenn es eine Sache gibt, die höhere und niedrigere Programmiersprachen unterscheidet, ist es wahrscheinlich die Speicherverwaltung. Assembler-Programmierer müssen ständig über Speicher, Adressen, Layout von Programmen und Datenstrukturen nachdenken. Assembler und Linker bieten Hilfe, die effektiv genutzt werden muss. Aus diesem Grund werden in diesem Kapitel einige weitere Grundlagen der ARM-Architektur und die Funktionsweise der Toolchain erläutert.

Adressraum

In den bisherigen Beispielen wurden Adressen für Peripherieregisterzugriffe und Sprunganweisungen verwendet, ohne wirklich zu erklären, was sie bedeuten. Es ist also an der Zeit, dies nachzuholen. Für den Zugriff auf

Peripherieregister und Speicherplätze in beliebigen Speichertypen (RAM, Flash, EEPROM…) wird eine Adresse benötigt, die den gewünschten Speicherplatz kennzeichnet. Auf den meisten Plattformen sind Adressen einfach Ganzzahlen ohne Vorzeichen. Die Menge aller möglichen Adressen, auf die einheitlich zugegriffen werden kann, wird als "Adressraum" bezeichnet. Einige Plattformen wie AVR haben mehrere Adressräume (für Flash, EEPROM und RAM + Peripherie), in denen auf jeden Speicher auf unterschiedliche Weise zugegriffen werden muss und der Programmierer wissen muss, zu welchem Adressraum eine Adresse gehört - z. Alle drei Speichertypen haben einen Speicherplatz mit der Adresse 123.

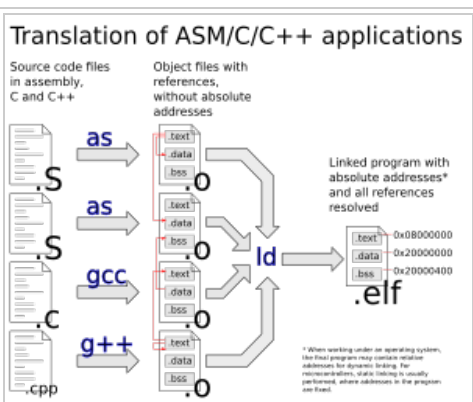
Die ARM-Architektur verwendet jedoch nur einen einzigen großen Adressraum, in dem Adressen vorzeichenlose 32-Bit-Ganzzahlen im Bereich von 0 bis 4294967295 sind. Jede Adresse bezieht sich auf ein Byte mit 8 Bits. Der Adressraum ist in mehrere kleinere Bereiche unterteilt, die sich jeweils auf einen bestimmten Speichertyp beziehen. Für den STM32F103 ist dies im Datenblatt in Kapitel 4 dokumentiert. Auf alle Adressen in allen Speichertypen wird auf die gleiche Weise zugegriffen - direkt über die Anweisungen "ldr" und "str" oder durch Ausführen von Code von einem bestimmten Ort aus, der dies kann erreicht werden, indem mit dem Befehl "b" zur jeweiligen Adresse gesprungen wird. Dies ermöglicht auch die Ausführung aus dem RAM - führen Sie einfach einen Sprung zu einer Adresse durch, die auf einen im RAM befindlichen Code verweist. Beachten Sie, dass zwischen den einzelnen Bereichen im Adressraum große Lücken bestehen. Der Versuch, auf diese zuzugreifen, führt normalerweise zu einem Absturz.

Während die Adressen der Peripherie vom Hersteller festgelegt und definiert sind, kann das Layout des Programmcodes und der Daten im Speicher vom Programmierer ziemlich frei eingestellt werden. Bisher haben die Beispielprogramme den Inhalt des Flash-Speichers linear definiert, indem sie die Anweisungen in der Reihenfolge aufgelistet haben, in der sie im Flash-Speicher erscheinen sollen. Wenn Sie jedoch mehrere Assembly-Quelldateien in ein Programm übersetzen, ist die Reihenfolge, in der der Inhalt dieser Dateien im endgültigen Programm angezeigt wird, nicht von vornherein festgelegt. Auch wenn im letzten Beispiel die Speicherblöcke für RAM *vor* dem Code definiert wurden, steht der Code tatsächlich im Adressraum an erster Stelle. Was all diese Arbeit macht, ist der Linker.

Der Linker

In der Regel ist der Linker der letzte Schritt bei der Übersetzung von Quellcode in ein verwendbares Programm. Er wird oft übersehen, manchmal missverstanden, ist aber ein wichtiges und nützliches Werkzeug, wenn er richtig angewendet wird. Viele Einführungen in die Programmierung gehen darauf ein, um die Funktionsweise im Detail zu erläutern. Wie in jedem Handel erfordert auch die Embedded-Entwicklung die Beherrschung der Tools! Ein gutes Verständnis des Linkers kann Zeit sparen, um seltsame Fehler zu beheben, und es Ihnen ermöglichen, einige weniger häufige Anwendungsfälle zu implementieren, z. B. die Verwendung mehrerer RAM-Blöcke in einigen Mikrocontrollern, das Ausführen von Code aus dem RAM oder das Definieren komplexer Speicherlayouts, wie dies manchmal von RTOS erforderlich ist.

Sie haben bereits einen Linker verwendet - den Befehl `arm-none-eabi-ld` der GNU-Linker, der mit der GNU-Toolchain geliefert wird. Bisher wurde für jedes Programm nur eine Assembly-Quelldatei übersetzt. Um ein größeres Programm zu übersetzen, das aus drei Assembly-Dateien "file1.S", "file2.S" und "file3.s" besteht, würde der Assembler dreimal aufgerufen, um drei Objektcode-Dateien "file1.o", "file2" zu erzeugen .o ”und“ file3.o ”. Der Linker würde dann aufgerufen, um alle drei in einer einzigen Ausgabedatei zu kombinieren.



Übersetzung nativer Anwendungen mit Assembler, Compiler und Linker

Beim Übersetzen einer dieser Assembly-Dateien weiß der Assembler nicht, ob die anderen Dateien vorhanden sind. Daher kann es nicht wissen, ob der Inhalt einer anderen Datei vor der aktuell verarbeiteten Datei im Flash-Speicher landet, und es kann auch nicht die endgültige Position des von ihr ausgegebenen und im Objekt platzierten Maschinencodes im Flash-Speicher ermittelt werden Datei (Endung .o). Dies bedeutet, dass die Objektdatei keine absoluten Adressen enthält (mit Ausnahme derjenigen von Peripherieregistern, da diese explizit angegeben wurden). Wenn zum Beispiel die Adresse der RAM-Datenblöcke ("ldr r0, = var1") geladen wird, kennt der Assembler die Adresse nicht, sondern nur der Linker. Daher fügt der Assembler einen Platzhalter in die Objektdatei ein, der vom Linker überschrieben wird. Ein Sprung ("b" -Anweisung) zu einer in einer anderen Assembly-Datei definierten Bezeichnung funktioniert ähnlich. Der Assembler verwendet einen Platzhalter für die Adresse. Für die Sprunganweisungen, die wir in derselben Datei (z. B. „b BlinkLoop“) verwendet haben, ist kein Platzhalter erforderlich, da der Assembler den Abstand zwischen Etikett und Anweisung berechnen und den relativen Sprung selbst generieren kann. Wenn sich das Ziel jedoch in einem anderen Abschnitt befindet (siehe unten), ist dies nicht möglich und ein Platzhalter ist erforderlich. Da der Inhalt von Objektdateien keine feste Adresse hat und vom Linker verschoben werden kann, werden diese Dateien als verschiebbar bezeichnet.

Auf Unix-Systemen (einschließlich Linux) wird das ELF-Format (Executable and Linkable Format) sowohl für

Objektdateien als auch für ausführbare Programmdateien verwendet. Dieses Format wird auch von ARM und der GNU ARM-Toolchain verwendet. Da es ursprünglich für die Verwendung mit Betriebssystemen gedacht war, können einige seiner Konzepte den eingebetteten Anwendungsfall nicht perfekt abbilden. Die vom Assembler und Linker erstellten Objektdateien (.o) sowie das endgültige Programm (normalerweise keine Endung, aber in eingebetteten Kontexten und auch in den obigen Beispielbefehlen wird .elf verwendet) sind alle im ELF-Format. Die Spezifikation von ELF für ARM finden Sie hier, und die generische Spezifikation für ELF, auf der die ARM-ELF-Variante basiert, kann sein gefunden hier.

ELF-Dateien sind in Abschnitte unterteilt. Jeder Abschnitt kann Code, Daten, Debug-Informationen (von GDB verwendet) und andere Dinge enthalten. In einer Objektdatei haben die Abschnitte keine feste Adresse. In der endgültigen Programmdatei haben sie eine. Abschnitte haben auch verschiedene Attribute, die angeben, ob es sich bei ihrem Inhalt um ausführbaren Code oder um Daten handelt, ob sie schreibgeschützt sind und ob ihr Speicher zugewiesen werden soll. Der Linker kombiniert und ordnet die Abschnitte aus den Objektdateien ("Eingabeabschnitte") neu und ordnet sie in Abschnitten in der endgültigen Programmdatei ("Ausgabeabschnitte") unter Zuweisung absoluter Adressen an.

Ein weiterer wichtiger Aspekt sind Symbole. Ein Symbol definiert einen Namen für eine Adresse. Die Adresse eines Symbols kann als absolute Zahl (z. B. 0x08000130) oder als Versatz relativ zum Beginn eines Abschnitts (z. B. "Startadresse des Abschnitts .text plus 0x130") definiert werden. Beschriftungen, die im Assembly-Quellcode definiert sind, definieren Symbole in der resultierenden Objektdatei. Beispielsweise führt die im letzten Beispiel definierte Bezeichnung "var1" zu einem Symbol "var1" in der Datei "prog1.o", dessen Adresse auf den Anfang von ".data" festgelegt ist. Das Symbol "var" wird ähnlich definiert, jedoch mit einem Versatz von 4. Nach dem Verknüpfungsvorgang enthält die Datei "prog1.elf" einen Abschnitt ".data" mit der absoluten Adresse 0x20000000, und daher die Abschnitte "var1" und "var2". Symbole erhalten auch absolute Adressen.

Wie bereits erwähnt, fügt der Assembler Platzhalter in die Objektdateien ein, wenn er die Adresse von etwas nicht kennt. In ELF-Dateien werden dort Platzhalter als „Umsiedlungseinträge“ bezeichnet und verweisen mit ihrem Namen auf Symbole. Wenn der Linker einen solchen Umsiedlungseintrag in einer seiner Eingabedateien sieht, sucht er in den Eingabedateien nach einem Symbol mit einem passenden Namen und gibt seine Adresse ein. Wenn kein Symbol mit diesem Namen gefunden wurde, wird dieser gefürchtete Fehler ausgegeben:

```
(. text + 0x132): undefinierter Verweis auf `Foo '
```

Google findet fast eine Million Ergebnisse für diese Nachricht. Wenn Sie jedoch wissen, wie der Linker funktioniert, können Sie diese leicht verstehen und lösen. Da das Symbol in keiner Objektdatei gefunden wurde, stellen Sie sicher, dass es richtig geschrieben ist und die Objektdatei, in der es sich befindet tatsächlich an den Linker gefüttert.

Linker Skripte

Ein Linkerskript ist eine Textdatei, die in einer linkerspezifischen Sprache geschrieben ist und steuert, wie der Linker Eingabeabschnitte Ausgabeabschnitten zuordnet. Das Beispielprojekt hat noch kein explizites angegeben, sodass der Linker ein integriertes Standardprojekt verwenden kann. Dies hat bisher funktioniert, führt jedoch zu einer leicht vertauschten Programmdatei (ungeeignete Symbole) und hat einige andere Nachteile. Daher ist es an der Zeit, die Dinge richtig zu machen und ein Linker-Skript zu schreiben. Linker-Skripte werden normalerweise nicht pro Projekt erstellt, sondern normalerweise vom Hersteller des Mikrocontrollers bereitgestellt, um das Speicherlayout eines bestimmten Controllers anzupassen. Um zu lernen, wie sie funktionieren, folgt eine kurze Einführung in das Schreiben. Die vollständige Dokumentation finden Sie hier.

Es ist üblich, das Linker-Skript nach dem Controller zu benennen, für den sie vorgesehen sind. Erstellen Sie daher eine Textdatei mit den folgenden Inhalten: "stm32f103rb.ld" oder "stm32f103c8.ld"

```
ERINNERUNG {
FLASH: ORIGIN = 0x8000000, LÄNGE = 128 KB
SRAM: ORIGIN = 0x20000000, LENGTH = 20K
}

ABSCHNITTE {
.text: {
*(.Text)
}> FLASH

.data (NOLOAD): {
*(.Daten)
}> SRAM
}
```

Beispielname: "LinkerScriptSimple"

Dies ist das minimale brauchbare Linker-Skript für einen Mikrocontroller. Wenn Sie einen STM32F103C8

verwenden, ersetzen Sie die 128K durch 64K. Die Zeilen innerhalb des Blocks „MEMORY“ definieren die verfügbaren Speicherbereiche auf Ihrem Mikrocontroller, indem Sie deren Startadresse und Größe im Adressraum angeben. Die Namen "FLASH" und "SRAM" können beliebig gewählt werden, da sie keine besondere Bedeutung haben. Diese Speicherdefinition hat außerhalb des Linker-Skripts keine Bedeutung, da sie nur ein interner Helfer zum Schreiben des Skripts ist. Es kann sogar weggelassen und durch einige manuelle Adressberechnungen ersetzt werden.

Der interessante Teil findet im Befehl "SECTIONS" statt. Jeder Untereintrag definiert einen Ausgabeabschnitt, der in der endgültigen Programmdatei endet. Diese können beliebig benannt werden, üblicherweise werden jedoch die Namen ".text" und ".data" für ausführbaren Code bzw. Datenspeicher verwendet. Die Stern-Ausdrücke „* (. Text)“ und „(*. Data)“ weisen den Linker an, den Inhalt der Eingabeabschnitte „. Text“ und „. Data“ an dieser Stelle im Ausgabeabschnitt abzulegen. In diesem Fall sind die Namen für die Eingabeabschnitte und Ausgabeabschnitte identisch. Die Namen der Eingabeabschnitte ".data", ".text" (und einige mehr) werden standardmäßig vom Assembler und den C- und C ++ - Compilern verwendet. Obwohl sie geändert werden können, ist es am besten, sie beizubehalten. Sie können die Ausgabeabschnitte jedoch beliebig benennen, zum Beispiel:

```
ABSCHNITTE {
.FlashText: {
*(.Text)
}> FLASH

.RamData (NOLOAD): {
*(.Daten)
}> SRAM
}
```

Die Befehle "> FLASH" und "> SRAM" weisen den Linker an, die Adresse der Ausgabeabschnitte gemäß der obigen Speicherdeklaration zu berechnen: Der erste Ausgabeabschnitt mit einem Befehl "> FLASH" endet an der Adresse 0x8000000, der nächste mit "> FLASH" direkt nach diesem Abschnitt und so weiter. Das "> SRAM" arbeitet genauso mit der Startadresse "0x20000000". Das Attribut "NOLOAD" ändert das Verhalten des Linkers nicht, markiert jedoch den entsprechenden Ausgabeabschnitt als "nicht ladbar", sodass OpenOCD und GDB nicht versuchen, ihn in den RAM zu schreiben - das Programm muss sich darum kümmern, RAM-Daten zu initialisieren sowieso, wenn Sie allein laufen.

Verwenden Sie die Option -T, um den Dateinamen des Linkerskripts anzugeben:

```
arm-none-eabi-ld prog1.o -o prog1.elf -T stm32f103rb.ld
```

Die `-Tdata </code> und <code> -Ttext` werden nicht mehr benötigt, da die Adressen jetzt im Linker-Skript definiert sind.

Da das Linker-Skript die Größe der Speicherbereiche definiert, kann der Linker Sie jetzt warnen, wenn Ihr Programm zu viel Speicher verbraucht (entweder Flash oder RAM):

```
 arm-none-eabi-ld: Abschnitt prog1.elf `.text 'passt nicht in Region` FLASH'
arm-none-eabi-ld: Region `FLASH 'ist um 69244 Bytes
```

übergelaufen

Speicherblöcke reservieren

Die Verwendung des Prozessors wird später erläutert. Sie können jedoch bereits das Linker-Skript verwenden, um einen Speicherblock zuzuweisen. Es ist am besten, Speicher für den Stack am Anfang des SRAM zu reservieren. Stellen Sie dies also vor den Befehl "* (. Data)":

```
. =. + 0x400;
```

In einem Linker-Skript bezieht sich der Punkt „.“ Auf die aktuelle Adresse in der Ausgabedatei. Aus diesem Grund erhöht dieser Befehl die Adresse um 0x400 und lässt einen "leeren" Block dieser Größe übrig. Der Eingabebereich ".data" befindet sich danach unter der Adresse 0x20000400.

Definieren von Symbolen in Linkerskripten

Wie bereits erwähnt, benötigt der Controller eine bestimmte Datenstruktur, die als "Vektortabelle" bezeichnet wird und sich ganz am Anfang des Flash-Speichers befindet. Es ist in der Assembler-Quelldatei definiert:

```
.word 0x20000400
.word 0x080000ed
.space 0xe4
```

Die Anweisung ".word" weist den Assembler an, die angegebene 32-Bit-Nummer auszugeben. Genau wie Prozessoranweisungen werden diese Nummern in den aktuellen Abschnitt eingefügt (standardmäßig .text, .data, falls angegeben) und landen daher im Flash-Speicher. Die erste 32-Bit-Nummer, die die ersten 4 Bytes im Flash-Speicher belegt, ist der Anfangswert des Stapelzeigers, der später erläutert wird. Diese Nummer sollte gleich der Adresse des ersten Bytes *nach* dem Speicherblock sein, der für den Stapel reserviert wurde. Der reservierte Block beginnt an der Adresse 0x20000000 und hat die Größe 0x400, sodass die richtige Nummer 0x20000400 lautet. Wenn jedoch die Größe des reservierten Blocks im Linkerskript geändert wurde, muss auch die obige Fertigungslinie angepasst werden. Um Inkonsistenzen zu vermeiden und alles, was mit dem Speicherlayout zu tun hat, zentral im Linker-Skript verwalten zu können, ist es wünschenswert, die Nummer in der Assembly-Quelldatei durch einen Symbolausdruck zu ersetzen. Dazu definieren Sie im Linker-Skript ein Symbol:

```
.data (NOLOAD): {
. =. + 0x400;
_StackEnd =.;
*(.Daten)
}> SRAM
```

Beispielname: „LinkerScriptSymbols“

Dies definiert ein Symbol "_StackEnd" mit dem Wert ".", Der aktuellen Adresse, die zu diesem Zeitpunkt 0x20000400 lautet. In der Assembly-Quelldatei können Sie jetzt die Nummer durch das Symbol ersetzen:

```
.word _StackEnd
```

Der Assembler fügt einen Platzhalter in die Objektdatei ein, den der Linker mit dem Wert 0x20000400 überschreibt. Durch diese Änderung wird die Ausgabedatei nicht geändert, es wird jedoch vermieden, absolute Adressen in Quelldateien einzufügen. Der Name "_StackEnd" wurde willkürlich gewählt; Da Namen, die mit einem Unterstrich und einem Großbuchstaben beginnen, in C- und C ++ - Programmen möglicherweise nicht verwendet werden, besteht keine Möglichkeit eines Konflikts, wenn später eine C / C ++ - Quelle hinzugefügt wird. In der Regel werden alle Symbole, die Teil der Laufzeitumgebung sind und für C / C ++ - Code "unsichtbar" sein sollten, auf diese Weise benannt. Die gleiche Regel gilt für Namen, die mit zwei Unterstrichen beginnen.

Der zweite Eintrag der Vektortabelle ist die Adresse des allerersten Befehls, der nach dem Zurücksetzen ausgeführt werden soll. Derzeit ist die Adresse als erste Adresse nach der Vektortabelle fest codiert. Wenn Sie vor dieser ersten Anweisung einen anderen Code einfügen möchten, muss diese Nummer geändert werden. Dies ist offensichtlich unpraktisch und daher sollte die Nummer auch durch ein Etikett ersetzt werden. Da der Code, der beim Zurücksetzen ausgeführt wird, gemeinhin als "Reset-Handler" bezeichnet wird, definieren Sie ihn folgendermaßen:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.word _StackEnd
.word Reset_Handler
.space 0xe4

.type Reset_Handler,% Funktion
Reset_Handler:

@ Code hier eingeben
```

Die Direktive ".type" teilt dem Assembler mit, dass sich die Bezeichnung auf ausführbaren Code bezieht. Die genaue Bedeutung davon wird später erläutert. Lassen Sie die .space-Direktive vorerst in Ruhe.

Absolute Abschnittsplatzierung

Die Vektortabelle muss sich am Anfang des Flash-Speichers befinden, und die Beispiele haben sich darauf verlassen, dass der Assembler die ersten Dinge aus der Quelldatei zuerst in den Flash-Speicher legt. Dies funktioniert nicht mehr, wenn Sie mehrere Quelldateien verwenden. Mit dem Linker-Skript können Sie sicherstellen, dass sich die Vektortabelle immer am Anfang des Flash-Speichers befindet. Dazu müssen Sie zuerst die Vektortabelle vom Rest des Codes trennen, damit der Linker speziell damit umgehen kann. Dazu platzieren Sie die Vektortabelle in einem eigenen Abschnitt:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.section .VectorTable, "a"
.word _StackEnd
.word Reset_Handler
.space 0xe4

.Text
.type Reset_Handler,% Funktion
Reset_Handler:
```

Beispielname: "LinkerScriptAbsolutePlacement"

Die Direktive ".section" weist den Assembler an, die folgenden Daten in den benutzerdefinierten Abschnitt ".VectorTable" einzufügen. Das "a" -Flag markiert diesen Abschnitt als zuweisbar, damit der Linker ihm Speicher zuweisen kann. Um die Vektortabelle am Anfang des Flash-Speichers zu platzieren, definieren Sie einen neuen Ausgabeabschnitt im Linker-Skript:

```
ERINNERUNG {
FLASH: ORIGIN = 0x8000000, LÄNGE = 128 KB
SRAM: ORIGIN = 0x20000000, LENGTH = 20K
}

ABSCHNITTE {
.VectorTable: {
* (. VectorTable)
}> FLASH

.text: {
*(.Text)
}> FLASH

.data (NOLOAD): {
. =. + 0x400;
_StackEnd =.;
*(.Daten)
}> SRAM
}
```

Dadurch wird der Eingabeabschnitt .VectorTable in den gleichnamigen Ausgabeabschnitt eingefügt. Es ist auch möglich, es neben dem Code in .text einzufügen:

```
ERINNERUNG {
FLASH: ORIGIN = 0x8000000, LÄNGE = 128 KB
SRAM: ORIGIN = 0x20000000, LENGTH = 20K
}

ABSCHNITTE {
.text: {
* (. VectorTable)
*(.Text)
}> FLASH

.data (NOLOAD): {
. =. + 0x400;
_StackEnd =.;
*(.Daten)
}> SRAM
}
```

Obwohl beide Varianten dasselbe Flash-Image erzeugen, ist es in GDB etwas angenehmer, mit dem ersten zu arbeiten. Die modifizierte LED-Blinker-Anwendung sieht nun so aus:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen
```

```
.section .VectorTable, "a"
.word _StackEnd
.word Reset_Handler
.space 0xe4

.Text
.type Reset_Handler,% Funktion
Reset_Handler:

ldr r1 = 0x40021018
ldr r0, [r1]
orr r0, r0, # 4
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren

ldr r1 = 0x40010804
ldr r0, [r1]
und r0, # 0xfffffff0
oder r0, # 2
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2


ldr r0, = 0x40010810 @ Adresse von GPIOA_BSRR laden
ldr r1, = 0x100 @ Wert registrieren, um Pin auf High zu setzen
ldr r2, = 0x1000000 @ Wert registrieren, um Pin auf Low zu setzen
ldr r3, = 1000000 @ Iterationen für die Verzögerungsschleife

BlinkLoop:
str r1, [r0] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

mov r4, r3
delay1:
subs r4, # 1
bne delay1 @ Verzögerungsschleife durchlaufen

str r2, [r0] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

mov r4, r3
delay2:
subs r4, # 1
bne delay2 @ Verzögerungsschleife durchlaufen

b BlinkLoop
```

Programmstruktur

Da die Vektortabelle in der Regel für alle Projekte gleich ist, ist es praktisch, sie in eine separate Datei mit dem Namen "vectortable.S" zu verschieben:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.section .VectorTable, "a"
.word _StackEnd
.word Reset_Handler
.space 0xe4
```

Assemblieren und verknüpfen Sie diesen Quellcode mit zwei Assembler-Befehlen:

```
arm-none-eabi-as-g prog1.S -o prog1.o
arm-none-eabi-as-g vectortable.S -o vectortable.o
arm-none-eabi-ld prog1.o vectortable.o -o prog1.elf -T stm32f103rb.ld
```

Dies führt zu dem gefürchteten Fehler "undefinierte Referenz". Um dies zu vermeiden, verwenden Sie die Direktive ".global" in der Hauptquelldatei "prog1.S":

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen
```

```
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:
@ Code hier ...
```

Dadurch wird der Assembler angewiesen, das Symbol "Reset_Handler" global sichtbar zu machen, damit es aus anderen Dateien verwendet werden kann. Standardmäßig erstellt der Assembler für jedes Etikett ein "lokales" Symbol, das nicht aus anderen Quelldateien verwendet werden kann (wie "statisch" in C). Das Symbol befindet sich jedoch noch in der endgültigen Programmdatei - es kann zu Debugging-Zwecken verwendet werden.

Weitere Montagetechniken

Nachdem Sie das Projekt für die ordnungsgemäße Verwendung des Linkers eingerichtet haben, werden einige weitere Aspekte der Assembly-Programmierung eingeführt.

Befehlssatzstatus

Wie bereits erwähnt, unterstützen ARM-Anwendungsprozessoren sowohl die ARM-Befehlssätze T32 als auch A32 / A64 und können dynamisch zwischen ihnen wechseln. Dies kann verwendet werden, um zeitkritische Programmteile im schnelleren A32 / 64-Befehlssatz und weniger kritische Teile im T32-Befehlssatz "thumb" zu codieren, um Speicherplatz zu sparen. Tatsächlich kann die Reduzierung der Programmgröße auch die Leistung verbessern, da die Cache-Speicher möglicherweise effektiver werden.

Auch wenn die auf der ARMv7-M-Architektur basierenden Cortex-M-Mikrocontroller die A32 / A64-Befehlssätze nicht unterstützen, ist ein Teil der Schaltlogik noch vorhanden, sodass der Programmcode entsprechend funktionieren muss. Die Umschaltung zwischen den Befehlssätzen erfolgt beim Springen mit den Befehlen "bx" "Branch and Exchange" und "blx" "Branch with Link and Exchange". Da alle Befehle die Größe 2 oder 4 haben und der Code nur an geraden Adressen gespeichert werden darf, ist das niedrigste Bit der Adresse eines Befehls immer Null. Wenn ein Sprung mit "bx" oder "blx" ausgeführt wird, wird das niedrigste Bit der Zieladresse verwendet, um den Befehlssatz des Sprungziels anzuzeigen: Wenn das Bit "1" ist, erwartet der Prozessor, dass der Code T32 ist, ansonsten A32.

Eine weitere Besonderheit der Befehle "bx" und "blx" besteht darin, dass sie die Sprungzieladresse aus einem Register entnehmen, anstatt sie direkt im Befehl zu codieren. Dies wird als indirekter Sprung bezeichnet. Ein Beispiel für einen solchen Sprung ist:

```
ldr r0, = SomeLabel
bx r0
```

Solche indirekten Sprünge sind notwendig, wenn die Differenz der Sprungzieladresse und des Sprungbefehls zu groß ist, um in dem Befehl selbst für einen relativen Sprung codiert zu werden. Manchmal möchten Sie auch zu einer Adresse springen, die von einem anderen Teil des Programms übergeben wurde, z. geschieht in C / C ++ -Code, wenn Funktionszeiger oder virtuelle Funktionen verwendet werden.

In diesen Fällen müssen Sie sicherstellen, dass das niedrigste Bit der Adresse, die über ein Register an „bx / blx“ übergeben wird, das niedrigste Bit gesetzt hat, um anzuzeigen, dass der Zielcode T32 ist. Andernfalls stürzt der Code ab. Dies kann erreicht werden, indem der Assembler über die bereits erwähnte Direktive ".type" informiert wird, dass sich die Zielkennung auf Code (und nicht auf Daten) bezieht:

```
.type SomeLabel,% Funktion
SomeLabel:
@ Ein bisschen Code ...
```

Auf diese Weise wird das niedrigste Bit gesetzt, wenn Sie auf das Etikett verweisen, um dessen Adresse in ein Register zu laden. Tatsächlich ist die Verwendung von ".type" für alle Code-Labels eine gute Idee, auch wenn es keine Rolle spielt, wenn Sie nur über die Anweisung "b" (einschließlich der bedingten Variante) auf ein Label verweisen, das nicht das niedrigste Bit codiert und dies auch tut Versuchen Sie nicht, einen Befehlssatzwechsel durchzuführen.

Wie bereits gezeigt, gibt es einen anderen Fall, in dem das niedrigste Bit von Bedeutung ist: Wenn Sie die Adresse des Rücksetz-Handlers (und später die Funktionen des Ausnahmehandlers) in der Vektortabelle angeben, muss das Bit gesetzt werden, daher lautet die Anweisung „.type“ auch hier notwendig:

```
.type Reset_Handler,% Funktion
```

Wenn Sie Code für einen Cortex-A-Prozessor schreiben, verwenden Sie ".arm" anstelle von ".thumb", um Ihren

Code (oder leistungskritische Teile davon) als A32 zu codieren. Die Direktive ".type" würde ebenfalls verwendet, und der Assembler würde das niedrigste Bit in der Adresse löschen, um sicherzustellen, dass der Code als A32 ausgeführt wird. Zum Beispiel:

```
.cpu cortex-a8
.syntax vereinheitlicht

@ Kleiner aber langsamer Code hier
.Daumen

Typ Block1,% Funktion
Block 1:
ldr r0, = Block2
bx r0

@ Größerer aber schnellerer Code hier
.Arm

Typ Block2,% Funktion
Block2:
@ ...
```

Die Direktive „.code 32“ hat die gleiche Bedeutung wie „.arm“ und „.code 16“ die gleiche Bedeutung wie „.thumb“ (obwohl der Name leicht irreführend ist, da T32-Anweisungen auch 32-Bit-Anweisungen sein können). Es gibt auch ".type Label,% object", um zu deklarieren, dass sich ein Label auf Daten in Flash oder RAM bezieht. Dies ist optional, hilft jedoch bei der Arbeit mit Analysewerkzeugen (siehe unten).

Konstanten

Die vorhergehenden Beispiele enthalten viele Zahlen (insbesondere Adressen), deren Bedeutung für den Leser nicht offensichtlich ist - sogenannte "magische Zahlen". Da Code in der Regel um ein Vielfaches häufiger gelesen wird als geschrieben / geändert, ist die Lesbarkeit auch für Assembly-Code wichtig. Daher ist es üblich, Konstanten zu definieren, die Nummern wie Adressen Namen zuweisen, und Namen anstelle der Nummer direkt zu verwenden.

Der Assembler bietet tatsächlich keinen dedizierten Mechanismus zum Definieren von Konstanten. Stattdessen werden die zuvor eingeführten Symbole verwendet. Sie können ein Symbol auf eine der folgenden Arten definieren:

```
RCC_APB2ENR = 0x40021018
.set GPIOA_CRH, 0x40010804
.equ GPIOA_ODR, 0x4001080C
```

und benutze es dann anstelle der Nummer:

```
ldr r1, = RCC_APB2ENR
```

Wenn Sie (fast) alle Zahlen im Quellcode für den LED-Blinker durch Konstanten ersetzen, erhalten Sie einen Quellcode wie diesen:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

RCC_APB2ENR = 0x40021018
RCC_APB2ENR_IOPAEN = 4
GPIOA_CRH = 0x40010804

GPIOA_BSRR = 0x40010810
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000

GPIOx_CRx_GP_PP_10MHz = 1
GPIOx_CRx_GP_PP_2MHz = 2
GPIOx_CRx_GP_PP_50MHz = 3

GPIOx_CRx_GP_OD_10MHz = 1 | 4
GPIOx_CRx_GP_OD_2MHz = 2 | 4
```

```
GPIOx_CRx_GP_OD_50MHz = 3 | 4

GPIOx_CRx_AF_PP_10MHz = 1 | 8
GPIOx_CRx_AF_PP_2MHz = 2 | 8
GPIOx_CRx_AF_PP_50MHz = 3 | 8

GPIOx_CRx_AF_OD_10MHz = 1 | 4 | 8
GPIOx_CRx_AF_OD_2MHz = 2 | 4 | 8
GPIOx_CRx_AF_OD_50MHz = 3 | 4 | 8

GPIOx_CRx_IN_ANLG = 0
GPIOx_CRx_IN_FLOAT = 4
GPIOx_CRx_IN_PULL = 8


DelayLoopIterations = 1000000

.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:

ldr r1, = RCC_APB2ENR
ldr r0, [r1]
orr r0, r0, # RCC_APB2ENR_IOPAEN
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren

ldr r1, = GPIOA_CRH
ldr r0, [r1]
und r0, # 0xfffffff0
orr r0, # GPIOx_CRx_GP_PP_2MHz
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2


ldr r0, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldr r1, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen
ldr r2, = GPIOx_BSRR_BR8 @ Wert registrieren, um Pin auf Low zu setzen
ldr r3, = DelayLoopIterations @ Iterations für die Verzögerungsschleife

BlinkLoop:
str r1, [r0] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

mov r4, r3
delay1:
subs r4, # 1
bne delay1 @ Verzögerungsschleife durchlaufen

str r2, [r0] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

mov r4, r3
delay2:
subs r4, # 1
bne delay2 @ Verzögerungsschleife durchlaufen

b BlinkLoop
```

Beispielname: "BlinkConstants"

Dies ist viel lesbarer als zuvor. In der Tat können Sie die Kommentare sogar weglassen, da der Code selbstdokumentierender wird. Die Adressen der Peripherieregister werden einzeln definiert, die Bits für die GPIO-Register sind jedoch für jedes GPIO-Modul gleich, sodass die Namen ein „x“ enthalten, um anzuzeigen, dass sie für alle GPIO-Module gelten.

Die Register "CRL" / "CRH" erhalten eine Sonderbehandlung. Da die einzelnen Bits wenig direkte Bedeutung haben, wäre es sinnlos, sie zu benennen. Stattdessen werden 15 Symbole definiert, um die 15 möglichen Betriebsmodi pro Pin zu kennzeichnen (Kombinationen aus Eingang / Ausgang, Open-Drain vs. Push-Pull, Analog vs. Digital, Floating vs. Pull-Widerständen und Anstiegsrate des Ausgangstreibers). . Jedes der 15 Symbole hat einen 4-Bit-Wert, der in die entsprechenden 4 Bits des Registers geschrieben werden muss. Zum Konfigurieren von z. PA10 als Allzweck-Open-Drain mit 10 MHz Anstiegsgeschwindigkeit:

```
ldr r1, = GPIOA_CRH
ldr r0, [r1]
```

```
und r0, # 0xfffff0ff
orr r0, # (GPIOx_CRx_GP_OD_10MHz << 8)
str r0, [r1]
```

C-ähnliche arithmetische Operatoren können in konstanten Ausdrücken wie + - * / und bitweisen Operatoren wie | verwendet werden (oder), & (und), << (Linksverschiebung) und >> (Rechtsverschiebung). Beachten Sie, dass diese Berechnungen immer vom Assembler durchgeführt werden. Im Beispiel oder | wird verwendet, um Bitwerte zu kombinieren.

Da es sich bei diesen Konstanten tatsächlich um Symbole handelt, können sie mit Assembler-Beschriftungen kollidieren. Daher dürfen Sie kein Symbol mit demselben Namen wie eine Beschriftung definieren.

Eine andere Art von Konstanten sind Register-Aliase. Mit der Direktive ".req" können Sie einen Namen für ein Prozessorregister definieren:

```
MyData .req r7
ldr MyData, = 123
MyData hinzufügen, 3
```

Dies kann für große Baugruppenblöcke nützlich sein, bei denen die Bedeutung von Registerdaten nicht offensichtlich ist. Sie können damit auch Register neu zuweisen, ohne viele Codezeilen ändern zu müssen.

Der Stack

In der Informatik ist ein Stapel eine dynamische Datenstruktur, in der Daten flexibel hinzugefügt und entfernt werden können. Wie ein Stapel Bücher muss das letzte Element, das darauf gelegt wurde, zuerst entnommen und entfernt werden (LIFO-Struktur - Last In, First Out). Das Hinzufügen eines Elements wird normalerweise als "Push" bezeichnet und das Lesen und Entfernen von "Pop".

Viele Prozessorarchitekturen, einschließlich ARM, verfügen über eine Schaltung, um mit einer solchen Struktur effizient umzugehen. Wie die meisten anderen stellt ARM hierfür keinen dedizierten Speicherbereich zur Verfügung. Es vereinfacht lediglich die Verwendung eines Bereichs, den der Programmierer für diesen Zweck als Stapel reserviert hat. Daher muss ein Teil des SRAM für den Stapel reserviert werden.

Auf ARM speichert das Programm Prozessorregister auf dem Stapel, d. H. 32 Bit pro Element. Der Stapel wird häufig verwendet, wenn der Inhalt eines Registers später erneut benötigt wird, nachdem er durch eine komplexe Operation überschrieben wurde, die viele Register benötigt. Diese Zugriffe erfolgen immer paarweise:

- Eine Operation, die nach r0 schreibt
- '*Push*' (save) r0 zum Stack
- Eine Operation, die r0 überschreibt
- '*Pop*' (Restore) r0 vom Stack
- Verwenden Sie den Wert in r0, der dem ursprünglich zugewiesenen Wert entspricht

Die Anweisungen von ARM für den Zugriff auf den Stapel werden nicht überraschend als "Push" und "Pop" bezeichnet. Sie können jedes der Register r0-r12 und r14 speichern / wiederherstellen, zum Beispiel:

```
ldr r0 = 1000000
@ Benutze r0 ...
Drücken Sie {r0} @ Save value 1000000

@… Ein Code, der r0 überschreibt…

pop {r0} @ Wert 1000000 wiederherstellen
@ Weiter mit r0 ...
```

Es ist auch möglich, mehrere Register auf einmal zu speichern / wiederherzustellen:

```
ldr r0 = 1000000
ldr r1 = 1234567
@ Benutze r0 und r1 ...
Drücken Sie {r0, r1} @ Speichern Sie die Werte 1000000 und 1234567

@… Ein Code, der r0 und r1 überschreibt…

pop {r0, r2} @ Stelle 1000000 in r0 und 1234567 in r2 wieder her
@ Weiter mit r0 und r2 ...
```

Es spielt keine Rolle, in welches Register die Daten zurückgelesen werden - im vorherigen Beispiel wird der in r1 gespeicherte Wert in r2 wiederhergestellt. In größeren Anwendungen werden viele Store-Restore-Paare verschachtelt:

```
ldr r0 = 1000000
@ Benutze r0 ...
Drücken Sie {r0} @ Save value 1000000

@ Innerer Codeblock:

ldr r0 = 123
@ Benutze r0…

{r0} drücken @ Wert 123 speichern

@ Inner-Inner-Code-Block, der r0 überschreibt

pop {r0} @ Wert 123 wiederherstellen
@ Weiter mit r0 ...

pop {r0} @ Stellen Sie den Wert 1000000 in r0 wieder her

@ Benutze weiterhin r0…
```

Das "innere" Push-Pop-Paar arbeitet mit dem Wert 123, und das "äußere" Push-Pop-Paar arbeitet mit dem Wert 1000000. Angenommen, der Stack war zu Beginn leer, enthält er nach dem ersten "Push" 1000000 und beides 1000000 und 123 nach dem zweiten Druck. Nach dem ersten "Pop" enthält es wieder nur 1000000 und ist nach dem zweiten "Pop" leer.

Zu Beginn eines Push-Pop-Paars ist der aktuelle Inhalt des Stapels irrelevant - er kann leer sein oder viele Elemente enthalten. Nach dem „Pop“ wird der Stapel in seinen vorherigen Zustand zurückversetzt. Dadurch ist es möglich, Push-Pop-Paare (fast) beliebig zu verschachteln - nachdem ein inneres Push-Pop-Paar abgeschlossen wurde, befindet sich der Stack im selben Zustand wie vor der Eingabe des inneren Paares, also im "Pop" -Teil des äußeren Paar bemerkt nicht einmal, dass der Stack dazwischen manipuliert wurde. Aus diesem Grund ist es wichtig sicherzustellen, dass jeder „Push“ einen passenden „Pop“ hat und umgekehrt.

Wie bereits erwähnt, muss ein Speicherbereich für den Stack reserviert werden. Der Zugriff auf den Stapelspeicher wird über den Stapelzeiger (SP) verwaltet. Der Stapelzeiger befindet sich im Prozessorregister r13, und "sp" ist ein Alias dafür. Wie der Name schon sagt, enthält der Stapelzeiger eine 32-Bit-Speicheradresse - insbesondere die Adresse des ersten Bytes im Stapel, das gespeicherte Daten enthält.

Wenn Sie einen 32-Bit-Registerwert mit „push“ speichern, wird der Stapelzeiger zuerst um 4 dekrementiert, bevor der Wert an die neu berechnete Adresse geschrieben wird. Um einen Wert wiederherzustellen, wird die aktuell im Stapelzeiger gespeicherte Adresse aus dem Speicher gelesen, woraufhin der Stapelzeiger um 4 erhöht wird. Dies wird als "vollständig absteigender" Stapel bezeichnet (siehe Referenzhandbuch zur ARM-Architektur, Kapitel B1.5.6)). In ARMv7-A (Cortex-A) kann dieses Verhalten geändert werden. In ARMv7-M wird es jedoch von der Ausnahmebehandlungslogik vorgegeben, die später erläutert wird.

Dies hat zur Folge, dass der Stapelzeiger, wenn der Stapel leer ist, die Adresse des ersten Bytes '*nach*' dem Stapelspeicherbereich enthält. Wenn der Stack vollständig gefüllt ist, enthält er die Adresse des allerersten Bytes '*innerhalb*' des Stack-Speicherbereichs. Dies bedeutet, dass der Stapel nach unten wächst. Da der Stapel beim Programmstart leer ist, muss der Stapelzeiger daher auf die erste Adresse nach dem Speicherbereich initialisiert werden. Vor dem Ausführen des ersten Befehls lädt der Prozessor die ersten 4 Bytes aus dem Flash in den Stapelzeiger. Aus diesem Grund wurde "_StackEnd" definiert und verwendet, um die Adresse des ersten Bytes nach dem Stapelspeicherbereich in die ersten 4 Bytes des Flash zu platzieren.

Der Stapelzeiger muss immer ein Vielfaches von 4 sein (siehe Kapitel B5.1.3 im ARM Architecture Reference Manual). Es ist ein häufiger Fehler (der sogar in den Beispielprojekten von ST! Vorhanden ist), den Stapelzeiger auf die letzte Adresse innerhalb des Stapelspeicherbereichs zu initialisieren (z. B. 0x200003FF anstelle von 0x20000400), der nicht durch vier teilbar ist . Dies kann dazu führen, dass die Anwendung abstürzt oder sie „nur“ verlangsamt. Tatsächlich erfordert der ARM ABI, dass der Stapelzeiger für öffentliche Software ein Vielfaches von 8 ist Schnittstellen, was wichtig ist für z die Funktion "printf" C. Stellen Sie daher beim Aufrufen von externem Code sicher, dass der Stapelzeiger ein Vielfaches von 8 ist.

In den vorhergehenden Beispielen wurde der Stapelspeicherbereich mit einer Größe von 0x400, d. H. 1 KB, definiert. Die Auswahl einer geeigneten Stapelgröße ist für eine Anwendung von entscheidender Bedeutung. Wenn es zu klein ist, stürzt die Anwendung ab. Wenn es zu groß ist, wird Speicher verschwendet, der sonst verwendet werden könnte. Traditionell ist der Stapel so konfiguriert, dass er sich am "Ende" des verfügbaren Speichers befindet, z. 0x20005000 für den STM32F103. Wenn der Linker zu Beginn des Speichers beginnt, Speicher für Daten zuzuweisen (unter Verwendung von „.data“ in Assembly oder globalen / statischen Variablen in C), ist der Stapel so weit wie möglich von diesen regulären Daten entfernt, wodurch die Wahrscheinlichkeit

einer Kollision minimiert wird . Wenn der Stapel jedoch kontinuierlich wächst, zeigt der Stapelzeiger möglicherweise in den regulären Datenbereich (".data" oder C-Globals) oder in den Heap-Speicher (von "malloc" in C verwendet). In diesem Fall werden beim unbeaufsichtigten Schreiben in den Stapel einige der regulären Daten überschrieben. Dies kann zu allen Arten von schwer zu findenden Fehlern führen. Daher setzen die Beispielcodes den Stapelbereich an den "Anfang" des RAM und die regulären Daten danach. Wenn der Stapel zu groß wird, erreicht der Stapelzeiger Werte unter 0x20000000, und jeder Zugriff führt zu einem sofortigen Zugriff "Sauberer" Absturz. Es ist wahrscheinlich einfach, die Codestelle zu finden, die zu viel Stapelspeicher zuweist, und möglicherweise die Stapelgröße zu erhöhen. Die Verwendung der Memory Protection Unit (MPU) des Cortex-M3 ermöglicht noch ausgefeiltere Strategien, die in diesem Lernprogramm jedoch nicht behandelt werden.

Funktionsaufrufe

Viele Programmiersprachen verfügen über ein Funktionskonzept. Funktionen, auch als "Prozeduren" oder "Unterprogramme" bezeichnet, sind die grundlegendsten Bausteine größerer Anwendungen, und ihre korrekte Anwendung ist der Schlüssel für sauberen, wiederverwendbaren Code. Der Assembler kennt die Funktionen nicht direkt, daher müssen Sie sie selbst erstellen. Eine Funktion ist ein Codeblock (d. H. Eine Folge von Anweisungen), zu dem Sie springen können, der etwas Arbeit leistet und der dann zu der Stelle zurückspringt, von der aus der erste Sprung ausgeführt wurde. Diese Fähigkeit zum Zurückspringen ist der Hauptunterschied zu jedem anderen Block von Assembler-Code. Um dies zu verdeutlichen, wird ein solcher Sprung zu einer Funktion als "Aufruf" bezeichnet (wie beim "Aufrufen einer Funktion"). Die Stelle im Code, an der der Sprung zur Funktion gestartet wird, wird als "Anrufer" und die aufgerufene Funktion als "Angerufene" bezeichnet. Aus der Sicht des Aufrufers ähnelt das Aufrufen einer Funktion einer "benutzerdefinierten" Anweisung - sie führt eine Operation aus, nach der der Code des Aufrufers wie zuvor fortgesetzt wird. Um das Zurückspringen zu ermöglichen, muss die Adresse der Anweisung *next* nach derjenigen, die den Funktionsaufruf gestartet hat, gespeichert werden, damit die Funktion zu dieser Position zurückspringen kann (ohne die Funktion direkt erneut aufzurufen).

Dies geschieht über das Link Register (LR), das das Prozessorregister r14 ist. Funktionsaufrufe werden mit der Anweisung „bl“ ausgeführt. Dieser Befehl führt einen Sprung aus, ähnlich wie das bekannte "b", speichert jedoch auch die Adresse des nächsten Befehls in LR. Wenn die Funktion beendet ist, kehrt sie zum Anrufer zurück, indem sie zu der in LR gespeicherten Adresse springt. Wie bereits erwähnt, wird das Springen von einem Register zu einer Stelle ein indirekter Sprung genannt, der durch den "bx" -Befehl ausgeführt wird. Um von einer Funktion zurückzukehren, verwenden Sie "bx lr":

```
.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:

bl EnableClockGPIOA @ Call-Funktion zum Aktivieren der GPIOA-Peripherietaktung

@ Noch etwas Code ...
ldr r1, = GPIOA_CRH
ldr r0, [r1]
und r0, # 0xfffffff0
orr r0, # GPIOx_CRx_GP_PP_2MHz
str r0, [r1]


.type EnableClockGPIOA,% Funktion
EnableClockGPIOA:
ldr r1, = RCC_APB2ENR
ldr r0, [r1]
orr r0, r0, # RCC_APB2ENR_IOPAEN
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren
bx lr @ Zurück zum Anrufer
```

Hier wurde der Code zum Aktivieren der Uhr für GPIOA in eine Funktion gepackt. Um diese Uhr zu aktivieren, ist nur noch eine einzige Zeile erforderlich - "bl EnableClockGPIOA".

Beim Aufrufen einer Funktion stellt der Befehl "bl" automatisch sicher, dass das niedrigste Bit in LR so gesetzt wird, dass das nachfolgende "bx lr" aufgrund eines versuchten Befehlsgruppenwechsels nicht abstürzt, was bei Cortex-M nicht möglich ist. Wenn Sie eine Funktion indirekt aufrufen müssen, verwenden Sie "blx" mit einem Register und achten Sie darauf, dass das niedrigste Bit gesetzt ist, normalerweise über ".type YourFunction,% function". Normalerweise befindet sich der gesamte Code einer Anwendung in Funktionen, mit der möglichen Ausnahme des Reset_Handlers. Die Reihenfolge, in der Funktionen in den Quelldateien definiert sind, spielt keine Rolle, da der Linker immer automatisch die richtigen Adressen angibt. Wenn Sie Funktionen in separaten Quelldateien ablegen möchten, müssen Sie ".global FunctionName" verwenden, um sicherzustellen, dass das Symbol für andere Dateien sichtbar ist.

Verwenden des Stapels für Funktionen

In großen Anwendungen ist es üblich, dass Funktionen andere Funktionen tief verschachtelt aufrufen. Eine wie gezeigt implementierte Funktion kann dies jedoch nicht. Wenn Sie "bl" verwenden, wird das LR überschrieben, sodass die Rücksprungadresse der äußeren Funktion verloren geht und diese Funktion niemals zurückkehren kann. Die Lösung besteht darin, den Stack zu verwenden: Verwenden Sie am Anfang einer Funktion, die andere Funktionen aufruft, "push", um das LR zu speichern, und am Ende "pop", um es wiederherzustellen. Zum Beispiel könnte das Blink-Programm folgendermaßen umstrukturiert werden:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

RCC_APB2ENR = 0x40021018
RCC_APB2ENR_IOPAEN = 4
GPIOA_CRH = 0x40010804

GPIOA_BSRR = 0x40010810
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000

GPIOx_CRx_GP_PP_2MHz = 2

DelayLoopIterations = 1000000

.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:
bl EnableClockGPIOA
bl PA8 konfigurieren
ldr r5, = 5 @ Anzahl der LED-Blitze.
bl Blink
b.

Typ Blinken,% Funktion
Blinken:
drücke {lr}
ldr r0, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldr r1, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen
ldr r2, = GPIOx_BSRR_BR8 @ Wert registrieren, um Pin auf Low zu setzen
ldr r3, = DelayLoopIterations @ Iterations für die Verzögerungsschleife

BlinkLoop:
str r1, [r0] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

bl Verzögerung

str r2, [r0] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

bl Verzögerung

subs r5, # 1
bne BlinkLoop

Pop {lr}
bx lr

.type EnableClockGPIOA,% Funktion
EnableClockGPIOA:
ldr r1, = RCC_APB2ENR
ldr r0, [r1]
orr r0, r0, # RCC_APB2ENR_IOPAEN
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren
bx lr @ Zurück zum Anrufer


.type ConfigurePA8,% Funktion
ConfigurePA8:
ldr r1, = GPIOA_CRH
ldr r0, [r1]
und r0, # 0xfffffff0
orr r0, # GPIOx_CRx_GP_PP_2MHz
```

```
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2
bx lr

Typ Verzögerung,% Funktion
Verzögern:
mov r4, r3
DelayLoop:
subs r4, # 1
bne DelayLoop @ Iterate-Delay-Schleife
bx lr
```

Beispielname: "BlinkFunctions"

Der Reset_Handler ist jetzt viel hübscher. Es gibt jetzt Funktionen zum Aktivieren der GPIOA-Uhr, zum Konfigurieren von PA8 als Ausgang und eine, die die Ausführung verzögert, sodass die blinkende LED sichtbar ist. Die "Blink" -Funktion führt das Blinken aus, jedoch nur für 5 Blitze. Danach kehrt sie zurück (eine endlose Blinkschleife wäre nicht gut, um die Rückkehr zu demonstrieren). Wie Sie sehen, wird LR auf dem Stack gespeichert, damit „Blink“ weitere Funktionen aufrufen kann.

Die zwei Zeilen

```
Pop {lr}
bx lr
```

sind eigentlich länger als nötig. Es ist tatsächlich möglich, die Rücksendeadresse direkt vom Stapel in den Programmzähler PC zu laden:

```
Pop {pc}
```

Auf diese Weise wird die auf dem Stapel gespeicherte Rücksprungadresse direkt für den Rücksprung verwendet. Genauso können Sie mit "push" und "pop" andere Register speichern und wiederherstellen, während Ihre Funktion ausgeführt wird.

Konvention anrufen

Es ist eine schlechte Idee, ein großes Programm zu erstellen, wie im letzten Beispiel gezeigt. Für die Funktion „Verzögerung“ sind 1000000 erforderlich, um sich in r4 zu befinden. Die Funktion „Blinken“ setzt voraus, dass „Verzögerung“ r0-r2 und r5 nicht überschreibt und dass die Anzahl der Blitze über r5 angegeben wird. Solche Anforderungen können schnell zu einem komplexen Netz von Abhängigkeiten werden, das es unmöglich macht, größere Funktionen zu schreiben, die mehrere Unterfunktionen aufrufen oder etwas umstrukturieren. Daher ist es üblich, eine Aufrufkonvention zu verwenden, die definiert, welche Register eine Funktion überschreiben darf, welche sie behalten soll, wie sie den Stapel verwenden soll und wie Informationen an den Aufrufer zurückgegeben werden.

Wenn Sie eine gesamte Anwendung aus Ihrem eigenen Assemblycode erstellen, können Sie Ihre eigene Aufrufkonvention erfinden. Es ist jedoch immer eine gute Idee, vorhandene Standards zu verwenden: Das AAPCS definiert eine Aufrufkonvention für ARM. Diese Konvention wird auch von C- und C ++ - Compilern befolgt, sodass Ihr Code mit diesen automatisch kompatibel wird. Der Cortex-M-Interrupt-Mechanismus folgt ihm ebenfalls, was es schwierig machen würde, Code, der eine andere Konvention verwendet, an Interrupts anzupassen. Die Spezifikation der Aufrufkonvention ist recht komplex. Hier finden Sie eine kurze Zusammenfassung der Grundlagen:

- Funktionen dürfen nur die Register r0-3 und r12 ändern. Werden mehr Register benötigt, müssen diese mit dem Stack gespeichert und wiederhergestellt werden. Der APSR kann ebenfalls modifiziert werden.
- Der LR wird wie gezeigt für die Absenderadresse verwendet.
- Bei der Rückkehr (über „bx lr“) sollte der Stapel genau im gleichen Zustand sein wie beim Sprung zur Funktion (über „bl“).
- Die Register r0-r3 können verwendet werden, um zusätzliche Informationen an eine Funktion zu übergeben, die als Parameter bezeichnet wird, und die Funktion kann sie überschreiben.
- Über das Register r0 kann ein Ergebniswert an den Aufrufer zurückgegeben werden, der als Rückgabewert bezeichnet wird.

Dies bedeutet, dass Sie beim Aufrufen einer Funktion davon ausgehen müssen, dass die Register r0-r3 und r12 möglicherweise überschrieben werden, während die anderen ihre Werte beibehalten. Mit anderen Worten, die Register r0-r3 und r12 werden (wenn überhaupt) "außerhalb" der Funktion gespeichert ("Anrufer-Speichern"), und die Register r4-r11 werden (wenn überhaupt) "innerhalb" gespeichert 'die Funktion ("callee-save").

Eine Funktion, die keine anderen Funktionen aufruft, wird als „Blattfunktion“ bezeichnet (da es sich um ein Blatt im Aufrufbaum handelt). Wenn eine solche Funktion einfach ist, muss der Stack möglicherweise überhaupt nicht berührt werden, da der Rückgabewert nur in einem Register (LR) gespeichert wird und möglicherweise nur die Register r0-r3 und r12 überschreibt, die der Aufrufer sicherstellen kann keine wichtigen Daten enthalten. Dies macht kleine Funktionen effizient, da Registerzugriffe schneller sind als Speicherzugriffe, beispielsweise auf den Stapel.

Wenn alle Ihre Funktionen der Aufrufkonvention folgen, können Sie jede Funktion von überall aus aufrufen und sicher sein, was sie überschreibt, auch wenn sie viele andere Funktionen alleine aufruft. Die Umstrukturierung des LED-Blinkers könnte folgendermaßen aussehen:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

RCC_APB2ENR = 0x40021018
RCC_APB2ENR_IOPAEN = 4
GPIOA_CRH = 0x40010804

GPIOA_BSRR = 0x40010810
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000

GPIOx_CRx_GP_PP_2MHz = 2

DelayLoopIterations = 1000000

.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:
bl EnableClockGPIOA
bl PA8 konfigurieren
ldr r0, = 5
bl Blink
b.

Typ Blinken,% Funktion
Blinken:
drücke {r4-r7, lr}
ldr r4, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldr r5, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen
ldr r6, = GPIOx_BSRR_BR8 @ Wert registrieren, um Pin auf Low zu setzen
mov r7, r0 @ Anzahl der LED blinkt.

BlinkLoop:
    str r5, [r4] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

    ldr r0, = DelayLoopIterations @ Iterations für die Verzögerungsschleife
    bl Verzögerung

    str r6, [r4] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

    ldr r0, = DelayLoopIterations @ Iterations für die Verzögerungsschleife
    bl Verzögerung

    unter r7, # 1
    bne BlinkLoop

pop {r4-r7, pc}

.type EnableClockGPIOA,% Funktion
EnableClockGPIOA:
ldr r1, = RCC_APB2ENR
ldr r0, [r1]
orr r0, r0, # RCC_APB2ENR_IOPAEN
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren
bx lr @ Zurück zum Anrufer

.type ConfigurePA8,% Funktion
ConfigurePA8:
ldr r1, = GPIOA_CRH
ldr r0, [r1]
```

```
und r0, # 0xfffffff0
orr r0, # GPIOx_CRx_GP_PP_2MHz
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2
bx lr

@ Parameter: r0 = Anzahl der Iterationen
Typ Verzögerung,% Funktion
Verzögern:
DelayLoop:
subs r0, # 1
bne DelayLoop @ Iterate-Delay-Schleife
bx lr
```

Beispielname: "BlinkFunctionCallingConvention"

Die drei kleinen Funktionen am Ende verwenden nur die Register r0 und r1, die sie überschreiben können. Die Funktion „Verzögerung“ erwartet die Anzahl der Iterationen als Parameter in r0, die dann geändert werden. Daher füllt die Funktion "Blinken" vor jedem Aufruf von "Verzögerung" r0. Alternativ könnte "Verzögerung" einen festen Iterationszähler verwenden, d. H. Das "ldr" könnte in "Verzögerung" verschoben werden. Da die Funktion "Blinken" davon ausgehen muss, dass "Verzögerung" r0-r3 und r12 überschreibt, behält sie ihre eigenen Daten in r4-r7, die gemäß der Aufrufkonvention garantiert erhalten bleiben. Da "Blink" diese Register für die aufgerufene Funktion beibehalten muss, werden sie mit "push" und "pop" gespeichert und wiederhergestellt. Beachten Sie die verkürzte Syntax „r4-r7“ in der Anleitung. Die Anzahl der LED-Blitze wird als Parameter in r0 übergeben; Da dieses Register überschrieben wird, wird diese Nummer nach r7 verschoben.

Alternativ könnte „Blink“ die Konstanten jedes Mal neu laden, wenn sie in r1 / r2 verwendet werden, so dass nur ein Register (r4) gespeichert werden muss, da es zum Zählen der Anzahl der Blitze benötigt wird:

```
Typ Blinken,% Funktion
Blinken:
    drücke {r4, lr}

    mov r4, r0

    BlinkLoop:
   ldr r1, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
   ldr r2, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen
   str r2, [r1] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

   ldr r0, = DelayLoopIterations @ Iterations für die Verzögerungsschleife
   bl Verzögerung

   ldr r1, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
   ldr r2, = GPIOx_BSRR_BR8 @ Wert registrieren, um Pin auf Low zu setzen
   str r2, [r1] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

   ldr r0, = DelayLoopIterations @ Iterations für die Verzögerungsschleife
   bl Verzögerung

   subs r4, # 1
   bne BlinkLoop

    Pop {r4, pc}
```

Beispielname: "BlinkFunctionCallingConvention2"

Eine dritte Variante würde keines der callee-save-Register (r4-r11) verwenden und stattdessen nur r0 speichern, bevor die Funktion aufgerufen wird, und es nach Bedarf wiederherstellen

```
Typ Blinken,% Funktion
Blinken:
drücke {lr}

BlinkLoop:
{r0} drücken

ldr r1, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldr r2, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen
str r2, [r1] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

ldr r0, = DelayLoopIterations @ Iterations für die Verzögerungsschleife
```

```
bl Verzögerung

ldr r1, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldr r2, = GPIOx_BSRR_BR8 @ Wert registrieren, um Pin auf Low zu setzen
str r2, [r1] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

ldr r0, = DelayLoopIterations @ Iterations für die Verzögerungsschleife
bl Verzögerung

Pop {r0}
subs r0, # 1
bne BlinkLoop

Pop {pc}
```

Beispielname: "BlinkFunctionCallingConvention3"

Die häufigen Stapelzugriffe würden dies jedoch verlangsamen. Dokumentieren Sie immer die Bedeutung (und ggf. die Einheiten) von Parametern, z. über Kommentare.

Bedingte Ausführung

Wie erwähnt, können die bedingten Varianten des "b" -Befehls (z. B. "bne") verwendet werden, um bestimmte Codeblöcke nur dann auszuführen, wenn eine bestimmte Bedingung erfüllt ist. Zunächst werden weitere Möglichkeiten zum Formulieren von Bedingungen gezeigt. Als nächstes wird der ARM-Befehl "it" eingeführt, der die Ausführung kleiner Codeblöcke bedingt effizienter macht.

Bedingungen

Alle Bedingungen für die bedingte Ausführung hängen vom Ergebnis einer mathematischen Operation ab. Bei der Verwendung von Befehlen wie „Adds“, „Subs“, „Ands“ werden die Flags im APSR-Register abhängig vom Ergebnis aktualisiert. Diese werden dann von den bedingten Varianten von „b“ gelesen, um zu entscheiden, ob das tatsächlich ausgeführt werden soll springen.

Oft müssen zwei Zahlen ohne Berechnung verglichen werden. Dies kann mit der Anweisung "cmp" erfolgen, an die Sie zwei Register oder ein Register und ein Literal übergeben können:

```
cmp r0, # 42
cmp r0, r1
```

Der Befehl "cmp" ist "subs" sehr ähnlich - er subtrahiert den zweiten Operanden vom ersten, speichert das Ergebnis jedoch nirgendwo, d. H. Die Register behalten ihre Werte bei. Nur die Flags in der APSR werden entsprechend dem Ergebnis aktualisiert, genau wie bei "subs". Wenn beispielsweise beide Operanden gleich waren, ist das Ergebnis der Subtraktion Null und das Null-Flag wird gesetzt. So testen Sie, ob zwei Zahlen gleich sind:

```
cmp r0, # 42
beq TheAnswer

@ Dies wird ausgeführt, wenn r0 nicht 42 ist

Die Antwort:
@ Dies wird ausgeführt, wenn r0 42 ist
```

Die Anweisung "bne" ist das Gegenteil von "beq".

Der Befehl "tst" funktioniert ähnlich wie "cmp", aber anstatt zu subtrahieren, führen Sie eine bitweise "and" -Operation aus - wie der Befehl "ands", aber ohne das Ergebnis beizubehalten. Auf diese Weise können Sie testen, ob ein Bit in einem Register gesetzt ist:

```
tst r0, # 4
beq BitNotSet

@ Wird ausgeführt, wenn Bit 2 in r0 gesetzt ist

BitNotSet:
@ Wird ausgeführt, wenn Bit 2 in r0 nicht gesetzt ist
```

Ein nützlicherer Anwendungsfall für "tst" ist das zweimalige Übergeben desselben Registers. Wenn Sie „und“ zweimal auf denselben Wert anwenden, erhalten Sie dasselbe Ergebnis wie bei der Eingabe. In diesem Fall überprüft „tst“ die Eigenschaften der Eingabe (negativ / positiv, Null):

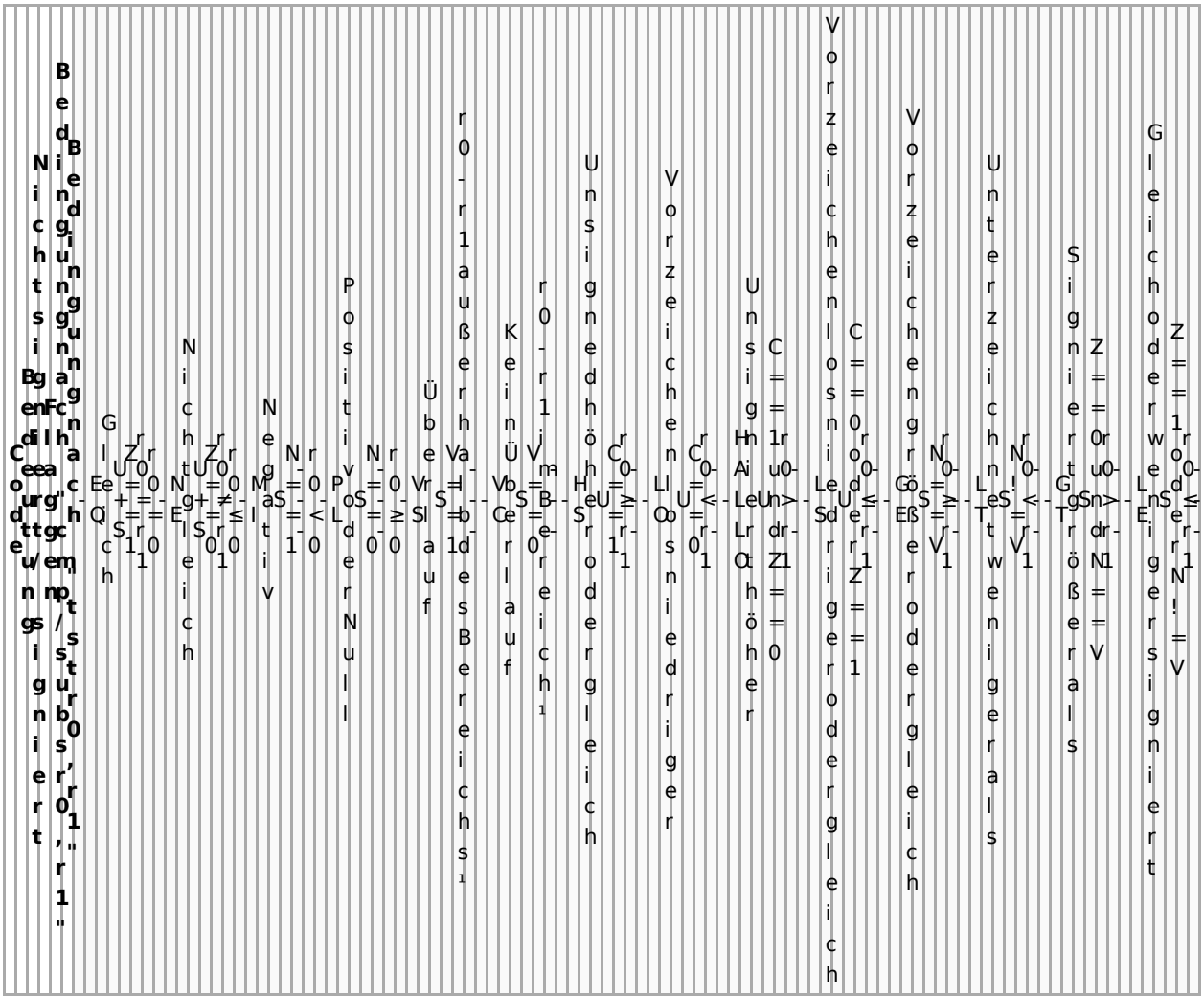
```
tst r0, r0
beq ValueZero

@ Dies wird ausgeführt, wenn r0 nicht Null ist

ValueZero:
@ Dies wird ausgeführt, wenn r0 Null ist
```

Es gibt auch die Anweisung "teq", die eine Exklusiv- oder Operation ausführt.

Wie bereits erwähnt, werden die Suffixe "eq" und "ne" als Bedingungscodes bezeichnet. ARM hat 14 davon, die definieren, wie die Flags in der APSR die Bedingung bilden. Die Details darüber, wie eine Subtraktion (durch "subs" oder "cmp") die Flags in der APSR setzt und wie ihre Interpretation durch die verschiedenen Bedingungscodes mit dem mathematischen Ergebnis korreliert, sind etwas kompliziert und hängen davon ab, wie das Komplementformat der 2 funktioniert und sich darauf stützt auf der Tatsache, dass das Subtrahieren durch Hinzufügen einer negierten Zahl funktioniert. Anstatt in alle Details einzutauchen, sollte eine Tabelle mit einer besseren Übersicht und einer praktischen Interpretation des Zustands hilfreicher sein:



1: Bereich, der die Zahlen von - (2 ^ 31) bis (2 ^ 31-1) einschließlich bedeutet

Um festzustellen, welchen Bedingungscode Sie benötigen, müssen Sie zunächst überlegen, ob die Nummer vorzeichenlos ist (Bereich 0 bis 2 ^ 32-1) oder das Zweierkomplement zur Darstellung vorzeichenbehafteter Zahlen verwendet (Bereich -2 ^ 31 bis 2 ^ 31-1). Ignorieren Sie alle Zeilen in der Tabelle mit dem falschen Format.

Wenn Sie zwei Zahlen vergleichen möchten, verwenden Sie die Anweisung "cmp" und suchen Sie in der Spalte

"cmp" der Tabelle nach der gewünschten Bedingung. Wenn Sie die Eigenschaften einer einzelnen Zahl testen möchten, verwenden Sie die Spalte „tst“. Verwenden Sie den Bedingungscode aus der ersten Spalte mit der bedingten Anweisung "b" ("bne", "beq", "bmi", "bpl", "bhs", ...) direkt nach dem entsprechenden "cmp" / "tst" " Anweisung.

Beachten Sie, dass alle Bedingungscodes einen entsprechenden Umkehrcode haben, der genau die negierte Bedeutung hat. Die meisten haben auch einen getauschten Partnercode, der dem Austauschen der Operanden für cmp entspricht.

Die IT-Anweisung

Das Springen ist ineffizient, so dass viele bedingte Sprünge Ihr Programm verlangsamen können. Die ARM-Architektur bietet die Möglichkeit, einige Anweisungen von der Bedingung abhängig zu machen, ohne dass ein Sprung über die Anweisung „it“ (if-then) erforderlich ist. Es wird anstelle eines bedingten Sprungs nach einer Anweisung verwendet, die die Flags ("cmp", "tst", "additives" ...) setzt, und benötigt auch einen Bedingungscode. Die nächste Anweisung direkt nach der wird dann nur ausgeführt, wenn die Bedingung erfüllt ist, und ansonsten übersprungen. Sie müssen den Bedingungscode wiederholen und zu dieser Anweisung hinzufügen. Dies dient nur dazu, den Code klarer zu gestalten und Verwirrung zu vermeiden.

```
ldr r4, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldr r5, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen

ldr r0, = 1 @ Ein Datum zum Vergleichen laden
ldr r1 = 2

cmp r0, r1 @ Vergleich durchführen

it hi @ Mache den nächsten Befehl von einer Bedingung abhängig
strhi r5, [r4] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen
```

Dies prüft, ob r0 höher als r1 ist (es ist nicht so) und setzt den Pin PA8 nur dann auf hoch, wenn diese Bedingung erfüllt ist. Auf diese Weise können bis zu 4 Anweisungen bedingt werden. für jedes muss ein zusätzliches "t" an die "it" -Anweisung angehängt werden:

```
cmp r0, r1 @ Vergleich durchführen

ittt hi @ Mache den nächsten Befehl von einer Bedingung abhängig
ldrhi r4, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldrhi r5, = GPIOx_BSRR_BS8 @ Registerwert, um den Pin auf hoch zu setzen
strhi r5, [r4] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen
```

Sie können auch Anweisungen hinzufügen, die ausgeführt werden, wenn die Bedingung *nicht* erfüllt wurde (wie ein 'else'-Fall in Hochsprachen), indem Sie an das' it 'ein' e 'anstelle von' t 'anhängen. Anweisung. Da das "t" in "it" festgelegt ist, wird der erste Befehl immer ausgeführt, wenn die Bedingung erfüllt ist. Nur die nächsten drei Anweisungen können entweder ein "Dann" -Fall ("t") oder ein "Sonst" -Fall ("e") sein. Sie müssen auch den invertierten Bedingungscode für die "else" -Anweisungen angeben:

```
ldr r4, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldr r5, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen
ldr r6, = GPIOx_BSRR_BR8 @ Wert registrieren, um Pin auf Low zu setzen

ldr r0, = 1 @ Ein Datum zum Vergleichen laden
ldr r1 = 2

cmp r0, r1 @ Vergleich durchführen

ite hi @ Mache die nächsten beiden Anweisungen abhängig (wenn-dann-sonst)
strhi r5, [r4] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen
strls r6, [r4] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen
```

Es gibt verschiedene Einschränkungen, welche Anweisungen in einem it-Block angezeigt werden dürfen. Am wichtigsten ist, dass Befehle, die die Flags setzen, hier verboten sind, ebenso wie der "b" -Befehl mit Ausnahme des letzten Befehls in einem "it" -Block. Das direkte Springen zu einer der bedingten Anweisungen ist ebenfalls verboten.

In T32-Code kann nur der bedingte "b" -Befehl einen Bedingungscode zusammen mit einer Operation codieren, sodass der "it" -Befehl bereitgestellt wird, um einen Befehl bedingt zu machen. In A32 enthalten die meisten Anweisungen einen Bedingungscode und können daher bedingt sein, und die "it" -Anweisung wird hier vom Assembler ignoriert. Sie können und sollten "es" in den für A32 bestimmten Code einfügen, da dies die

Kompatibilität mit T32 ermöglicht. Dies ist einer der Gründe, warum A32 zeiteffizienter und T32 platzsparender ist.

Bedingte Anweisungen erzeugen manchmal überraschend kompakte Programme. Zum Beispiel kann der euklidische Algorithmus zur Berechnung des größten gemeinsamen Divisors (gcd) zweier Zahlen in der ARM-Assembly folgendermaßen geschrieben werden:

```
gcd:
cmp r0, r1
ite gt
subgt r0, r0, r1
suble r1, r1, r0
bne gcd
```

Während das C-Äquivalent tatsächlich länger ist:

```
int gcd (int a, int b) {
während (a! = b) {
wenn (a> b)
a = a - b;
sonst
b = b - a;
}
return a;
}
```

Die Verwendung von bedingten Anweisungen ist auch [3]. Code schneller] als mit bedingten Sprüngen. Beachten Sie, dass der letzte Befehl „bne“ unabhängig vom Block „if-then“ ist. es wird nur direkt das Ergebnis von "cmp" verwendet.

8/16 Bit Arithmetik

Bisher hatten alle Zahlen 32 Bit. Insbesondere aus Platzgründen werden jedoch kleinere Zahlen mit 8 oder 16 Bit benötigt. Cortex-M3 bietet keine Anweisungen für die direkte Berechnung von 8- oder 16-Bit-Zahlen. Stattdessen muss eine solche Zahl nach dem Laden aus dem Speicher in ein Prozessorregister auf 32-Bit erweitert werden, damit die 32-Bit-Befehle ordnungsgemäß funktionieren. Beim Zurückspeichern des Ergebnisses wird nur das untere 8/16 Bit verwendet. Wenn 8/16-Bit-Überlaufverhalten erforderlich ist (dh Überlauf bei -128/127 für 8-Bit-Vorzeichen, 0/256 für 8-Bit-Vorzeichen, -32768/32767 für 16-Bit-Vorzeichen, 0/65536 für 16-Bit-Vorzeichen), müssen die Zahlen für Berechnungen verwendet werden nach jeder Berechnung abgeschnitten. Dies macht es tatsächlich etwas weniger effizient, mit kleineren Zahlen umzugehen.

Ein 16-Bit-Wert („Halbwort“) kann mit der Anweisung ldrh aus dem Speicher gelesen werden:

```
ldr r0, = SomeAddress
ldrh r1, [r0]
```

"Ldrh" lädt 16 Bit aus dem Speicher, schreibt sie in die unteren 16 Bit des Zielregisters (hier: r1) und setzt die oberen 16 Bit auf Null. Wenn der Wert signiert ist, muss er vorzeichenerweitert werden, damit er für 32-Bit-Berechnungen verwendet werden kann:

```
ldr r0, = SomeAddress
ldrh r1, [r0]
sxth r1, r1
```

Der Befehl "sxth" kopiert das Vorzeichenbit (d. H. Bit 15) in die oberen 16 Bits ("Vorzeichenerweiterung"); Dies stellt sicher, dass negative 16-Bit-Zahlen ihren Wert behalten, wenn sie als 32-Bit interpretiert werden. Der Befehl "ldrsh" kombiniert sowohl "ldrh" als auch "sxth". "Ldrb", "sxtb", "ldrsb" dienen zum Laden und Vorzeichen-Erweitern von 8-Bit-Werten bzw. der Kombination von beiden.

Um das 8/16-Bit-Überlaufverhalten nach einer mathematischen Operation zu simulieren, verwenden Sie uxtb / uxth für vorzeichenlose 8/16-Bit-Zahlen oder sxtb / sxth für vorzeichenbehaftete 8/16-Bit-Zahlen:

```
addiere r0, # 1
uxth r0, r0
```

Die Befehle "uxth" / "uxtb" kopieren die unteren 16/8 Bits eines Registers in ein anderes und setzen die oberen

16/24 Bits auf Null. Auf diese Weise wird, wenn r0 zuvor 65535 enthielt, das Ergebnis 0 anstelle von 65536 sein, nachdem "uxth" verwendet wurde.

Dies ist eine häufige Falle beim Codieren in C - wenn z. Beim Typ "uint16_t" für lokale Variablen, z. B. Schleifenzähler, wird implizit ein 16-Bit-Überlaufverhalten angefordert, das nach jeder Berechnung abgeschnitten werden muss, obwohl der Überlauf möglicherweise tatsächlich nie auftritt. Dies ist der Grund, warum z.B. uint16_fast_t sollte für lokale Variablen verwendet werden, da dies auf ARM 32 Bit ist, was schneller ist.

Ausrichtung

Für den Zugriff auf Daten im Speicher mit den Varianten "str" / "ldr" gelten bestimmte Adressbeschränkungen:

- Bei den Befehlen "ldrd" / "strd" / "ldm" / "stm", mit denen mehrere Register gleichzeitig geladen / gespeichert werden können, muss die Adresse immer ein Vielfaches von 4 sein. Ist dies nicht der Fall, stürzt das Programm ab .
- Für die Anweisungen "ldr" / "str" muss die Adresse ein Vielfaches von 4 sein, und für "strh" / "ldrh" muss ein Vielfaches von 2 sein. Ist dies nicht der Fall, hängt das Verhalten von der ARM-Version ab :
  - Auf ARMv6-M und früheren Versionen stürzt das Programm ab.
  - Auf ARMv7-M:
    - Wenn CCR.UNALIGN_TRP auf Null gesetzt ist (Standardeinstellung), ist der Zugriff langsam
    - Wenn das CCR.UNALIGN_TRP-Bit auf eins gesetzt ist, stürzt das Programm ab und emuliert das ARMv6-M-Verhalten

Für "strb" / "ldrb" gibt es keine derartigen Anforderungen.

Die Anzahl, bei der die Adresse ein Vielfaches von sein muss, wird als "Ausrichtung" bezeichnet (z. B. 2-Byte-Ausrichtung, 4-Byte-Ausrichtung, ...). Ein Zugriff mit einer Adresse, die ein Vielfaches von 2/4 ist, wie oben angegeben, wird als "ausgerichteter Zugriff" bezeichnet. andere werden als "nicht ausgerichteter Zugriff" bezeichnet (die langsam sind oder einen Absturz verursachen).

Auch wenn langsame Zugriffe akzeptabel sein mögen, ist es dennoch eine gute Idee, sicherzustellen, dass alle Zugriffe immer korrekt ausgerichtet sind, falls der Code auf eine ARM-Version oder ein Betriebssystem portiert wird, das dies erfordert. Die Adressen der Peripherieregister sind bereits korrekt ausgerichtet, sodass Sie sich keine Sorgen machen müssen. Wenn Sie Daten in den Arbeitsspeicher stellen, sollten Sie jedoch sicherstellen, dass die Adressen der einzelnen Elemente, auf die über eine der ldr-Varianten zugegriffen wird, richtig ausgerichtet sind. Wenn beispielsweise ein vorheriger Beispielcode wie folgt geändert wurde:

```
.Daten
var2:
.space 1 @ Reserve 1 Byte für Speicherblock "var2"
var1:
.space 4 @ Reserve 4 Bytes für den Speicherblock "var1"

.Text
@ Anleitung geh hier ...
```

Die Adresse von "var1" ist kein Vielfaches von 4, und ein Zugriff über "ldr" wäre nicht ausgerichtet. Dies könnte verbessert werden, indem ein Leerzeichen von 3 Bytes dazwischen hinzugefügt wird:

```
.Daten
var2:
.space 1 @ Reserve 1 Byte für Speicherblock "var2"
.space 3
var1:
.space 4 @ Reserve 4 Bytes für den Speicherblock "var1"

.Text
@ Anleitung geh hier ...
```

Dies würde erfordern, dass Sie alle anderen Dinge im Speicher berücksichtigen, die zuvor deklariert wurden. Dies ist insbesondere dann unpraktisch, wenn mehrere Assembly-Dateien verwendet werden. Daher bietet der Assembler die Direktive ".align" an:

```
.Daten
var2:
```

```
.space 1 @ Reserve 1 Byte für Speicherblock "var2"
.align 2
var1:
.space 4 @ Reserve 4 Bytes für den Speicherblock "var1"

.Text
@ Anleitung geh hier ...
```

Bei Verwendung von „.align X“ stellt der Assembler sicher, dass die nächste Adresse ein Vielfaches von 2 ^ X ist, in diesem Fall also ein Vielfaches von 2 ^ 2 = 4. Der Assembler fügt daher 0 bis 2 ^ X-1 Byte Speicherplatz ein. Der Abschnitt, der die Direktive in der Objektcodedatei enthält, wird ebenfalls markiert, um diese Ausrichtung zu erfordern, sodass der Linker sie automatisch an der entsprechenden Stelle im Adressraum platziert.

Versetzte Adressierung

Die verschiedenen "ldr" / "str" -Anweisungen können optional eine Berechnung der Adresse durchführen, bevor der Speicherzugriff ausgeführt wird. Was hier für "ldr" angezeigt wird, funktioniert für "str" und die Varianten für Halbwörter und Bytes gleichermaßen. Hierfür gibt es mehrere Varianten. Dieser erste fügt der Adresse einen festen Versatz hinzu, der innerhalb des Befehls selbst codiert ist:

```
ldr r0, [r1, # 8]
```

Dies addiert 8 zu r1 und verwendet das Ergebnis als Adresse für den Zugriff. Die Zahl kann auch negativ sein. Diese Variante ist nützlich, um auf Mitglieder eines heterogenen Containers zuzugreifen, der wie eine C-Struktur oder die Register in einem Peripheriemodul organisiert ist. Sie können zum Beispiel die Basisadresse eines Peripheriemoduls in ein Register laden und dann mithilfe der Offset-Adressierung auf die verschiedenen Register zugreifen, ohne jede Adresse einzeln laden zu müssen:

```
GPIOA = 0x40010800
GPIOx_CRH = 0x04
GPIOx_BSRR = 0x10

GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000

GPIOx_CRx_GP_PP_2MHz = 2

.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:
bl EnableClockGPIOA

ldr r1, = GPIOA

ldr r0, [r1, #GPIOx_CRH]
und r0, # 0xfffffff0
orr r0, # GPIOx_CRx_GP_PP_2MHz
str r0, [r1, #GPIOx_CRH] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2

ldr r0, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen
str r0, [r1, #GPIOx_BSRR] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

b.
```

Beispielname: „OffsetAddressing“

Auf diese Weise können Sie das wiederholte Laden ähnlicher Adressen vermeiden. Diese Variante kann auch die neu berechnete Adresse durch Anhängen eines "!" In das Adressregister zurückschreiben:

```
ldr r0, [r1, # 8]!
```

Dies addiert 8 zu r1, schreibt das Ergebnis in r1 und verwendet es auch als Adresse, von der 4 Bytes geladen und in r0 gespeichert werden. Die Variante

```
ldr r0, [r1], # 8
```

funktioniert genau umgekehrt - r1 wird als Adresse verwendet, von der die Daten geladen werden, und "r1 + 8" wird zurück in r1 geschrieben. Die nächste Variante fügt zwei Register hinzu, um die Speicheradresse zu erhalten:

```
ldr r0, [r1, r2]
```

Dadurch werden die Daten von der Adresse geladen, die mit „r1 + r2“ berechnet wurde. Das zweite Register (hier: r2) kann optional auch um eine feste Anzahl von Bits im Bereich 0-3 nach links verschoben werden:

```
ldr r0, [r1, r2, lsl # 2]
```

Dies verschiebt r2 um zwei Bits nach links (d. H. Multipliziert es mit 4), addiert es zu r1 und verwendet es als Adresse (r2 selbst wird nicht modifiziert).

Arrays durchlaufen

Der Offset-Adressierungsmechanismus eignet sich perfekt für die Iteration von Arrays. Dies könnte verwendet werden, um ein Array zu erstellen, das eine Sequenz von LED-Blitzen definiert, die von der LED-Blinker-Anwendung wiederholt wird. Ein solches Array würde die Dauer jedes Ein- und Ausschaltzyklus (wie an die Funktion "Verzögerung" übergeben) enthalten und im Flash-Speicher abgelegt werden:

```
.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:
bl EnableClockGPIOA
bl PA8 konfigurieren
bl Blink
b.

Typ Blinken,% Funktion
Blinken:
drücke {r4-r8, lr}
ldr r4, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldr r5, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen
ldr r6, = GPIOx_BSRR_BR8 @ Wert registrieren, um Pin auf Low zu setzen
ldr r7, = BlinkTable @ Adresse von "BlinkTable" in r7 verschieben
ldr r8, = BlinkTableEnd @ Adresse von "BlinkTableEnd" in r8 verschieben


BlinkLoop:
str r5, [r4] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

ldr r0, [r7], # 4 @ Iterationen der Ladeverzögerung von Tabelle und Inkrementadresse
bl Verzögerung

str r6, [r4] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

ldr r0, [r7], # 4 @ Iterationen der Ladeverzögerung von Tabelle und Inkrementadresse
bl Verzögerung

cmp r7, r8
blo BlinkLoop

pop {r4-r8, pc}

.align 2
.type BlinkTable,% Objekt
BlinkTable:
.Wort 1000000, 1000000, 1000000, 1000000, 1000000, 1000000
.Wort 2500000, 1000000, 2500000, 1000000, 2500000, 1000000
.Wort 1000000, 1000000, 1000000, 1000000, 1000000, 1000000
BlinkTableEnd:
```

Beispielname: "BlinkPattern"

Mit der Direktive „.word“ wird eine Folge von 32-Bit-Zahlen in den Flash-Speicher geschrieben. Die Bezeichnung „BlinkTable“ verweist auf die Startadresse des Arrays und „BlinkTableEnd“ auf die erste Adresse nach dem Array. Diese beiden Adressen werden vor der Schleife in Register geladen. Die Direktive ".align" wird verwendet, um sicherzustellen, dass die 32-Bit-Wörter an richtig ausgerichteten Adressen gespeichert werden. Innerhalb der

Schleife wird mit dem Befehl "ldr" ein 32-Bit-Wort aus dem Array geladen und an die Funktion "Delay" übergeben. Das r7-Register wird um 4 Bytes auf das nächste 32-Bit-Wort vorgerückt. Dies erfolgt zweimal für die Ein- und Ausschaltzeit. Am Ende der Schleife wird das Adressregister mit der Adresse von „BlinkTableEnd“ verglichen - bis diese Adresse erreicht ist, wird die Schleife fortgesetzt.

Eine andere Möglichkeit besteht darin, die Basisadresse des Arrays in einem Register zu belassen und ein anderes Register zu inkrementieren, das den Offset enthält:

```
Typ Blinken,% Funktion
Blinken:
drücke {r4-r9, lr}
ldr r4, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldr r5, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen
ldr r6, = GPIOx_BSRR_BR8 @ Wert registrieren, um Pin auf Low zu setzen
ldr r7, = BlinkTable @ Adresse von "BlinkTable" in r7 verschieben
ldr r8 = 0
ldr r9 = 18

BlinkLoop:
str r5, [r4] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

ldr r0, [r7, r8, lsl # 2] @ Lade Verzögerungsiterationen aus der Tabelle
füge r8, # 1 hinzu
bl Verzögerung

str r6, [r4] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

ldr r0, [r7, r8, lsl # 2] @ Lade Verzögerungsiterationen aus der Tabelle
füge r8, # 1 hinzu
bl Verzögerung

cmp r8, r9
blo BlinkLoop

pop {r4-r9, pc}
```

Beispielname: "BlinkPattern2"

Hier wird r8 in Schritten von 1 inkrementiert, um den Index im Array zu bezeichnen. Die “lsl” -Syntax für “ldr” wird verwendet, um r8 mit 4 zu multiplizieren (da jedes Wort 4 Byte groß ist) und es zu r7 hinzuzufügen, das die Basisadresse des Arrays enthält. Am Ende der Schleife wird r8 mit 18 verglichen, was der Anzahl der Einträge im Array entspricht. Diese Variante ist weniger effizient, da sie sowohl die Basisadresse als auch den Index in Registern halten und den Index bei jeder Iteration inkrementieren muss.

Literale Lasten

Unabhängig von der Architektur muss jeder Prozessor offensichtlich viel mit Adressen in seinem eigenen Adressraum arbeiten. ARM kann mit seinen 32-Bit-Adressen gut rechnen, aber es gibt einen Engpass: Der Befehlssatz selbst. Um mit einer beliebigen Adresse arbeiten zu können, muss sie zunächst in ein Prozessorregister geladen werden. ARM-Befehle haben jedoch nur eine Größe von 16 oder 32 Bit - nicht genügend Speicherplatz für eine beliebige 32-Bit-Zahl zuzüglich der Befehlscodierung. Das Zulassen noch größerer Anweisungen (z. B. 40 Bit) würde die Sache komplizieren, weshalb ARM stattdessen mehrere Tricks verwendet, um dieses Problem zu lösen, auf das hier eingegangen wird.

Die Syntax "ldr r0, = 1234" ermöglicht das Laden beliebiger 32-Bit-Zahlen, ist jedoch eigentlich keine Maschinencodeanweisung, sondern wird vom Assembler in eine übersetzt. In diesem Kapitel werden die tatsächlichen Anweisungen zum Laden von Sofortnummern erläutert.

Die "mov" -Anweisung

Die einfachste Methode zum Laden einer sofortigen Nummer in ein Register ist die Anweisung "mov":

```
mov r0, # 1234
```

Auf diese Weise können Sie eine beliebige 16-Bit-Nummer (0 bis 2 ^ 16-1) in ein Register laden. "Mov" enthält auch einige clevere Codierungen, mit denen Sie bestimmte häufig verwendete Muster laden können:

- Jede 32-Bit-Zahl, die aus einem Byte willkürlicher Bits (d. H. 8 benachbarte willkürliche Bits) an einer beliebigen Stelle und ansonsten aus Nullen besteht, z. 0x00000045, 0x00045000, 0x7f800000.

- Jede 32-Bit-Nummer, die aus demselben Byte besteht und 2 oder 4 Mal an festen Stellen wiederholt wird, wie in 0x23002300, 0x00230023, 0x23232323
- Das bitweise negierte Ergebnis eines dieser beiden Muster, z. 0xffffffba, 0xfffbafff, 0x807fffff oder 0xdcffdcff. Der Assembler verwendet hierfür die Anweisung "mvn", die identisch mit "mov" funktioniert, den Wert jedoch negiert.

Durch Angabe einer Zahl, die in eines dieser Muster fällt, verwendet der Assembler automatisch die entsprechende Codierung. Die ersten beiden Möglichkeiten zur Kodierung von Zahlen sind nicht nur mit "mov" verfügbar, sondern auch mit mehreren anderen mathematischen Befehlen, die einen unmittelbaren Wert erwarten: "add", "and", "bic", "cmn", "cmp", "eor" ”,” Mov ”,” mvn ”,” orn ”,” orr ”,” rsb ”,” sbc ”,” sub ”,” teq ”,” tst ”. Überprüfen Sie im ARM Architecture Reference Manual die Beschreibung der Anweisungen und achten Sie auf „ThumbExpandImm“, um festzustellen, ob die ersten beiden oben genannten Muster unterstützt werden.

Sie können den Befehl "mvn" auch direkt verwenden, z.

```
mov r0, # 0xf807ffff
mvn r0, # 0x07f80000
```

beide Zeilen sind identisch und schreiben die Nummer 0xf807ffff in r0.

Die Anweisung "movt"

Dies unterstützt zwar viele gängige Muster, lässt jedoch keine willkürlichen 32-Bit-Zahlen zu. Eine Möglichkeit, eine 32-Bit-Zahl zu laden, besteht darin, die Zahl in zwei 16-Bit-Hälften aufzuteilen und diese beiden Halbwörter mit "mov" und "movt" zu einem Register zu kombinieren:

```
mov r0, # 0xabcd
movt r0, # 0x1234
```

Der Befehl "movt" lädt die angegebene Nummer in die oberen 16 Bits des Registers. In diesem Beispiel wird also 0x1234abcd in r0 geladen. Die Reihenfolge ist wichtig, da "mov" die oberen 16 Bits mit Nullen überschreibt, "movt" jedoch die unteren 16 Bits beibehält. Wenn ein einzelnes "mov" nicht zu der gewünschten Nummer passt, ist die Kombination aus "mov" und "movt" die schnellste Möglichkeit, eine 32-Bit-Nummer zu laden. Da zwei 32-Bit-Befehle benötigt werden, belegt dies 8 Byte Programmspeicher. Wenn Sie die Adresse eines Symbols in ein Register laden möchten, müssen Sie den Assembler anweisen, es automatisch zu teilen. Dies kann erreicht werden, indem dem Symbol ": lower16:" oder ": upper16:" vorangestellt wird, z.

```
movw r0, #: lower16: GPIOA_BSRR
movt r0, #: upper16: GPIOA_BSRR
```

Beachten Sie, dass in diesem Fall "movw" angegeben werden muss, um den Assembler explizit anzuweisen, die "mov" -Variante zu verwenden, die 16-Bit-Zahlen akzeptiert (was ansonsten automatisch geschieht, wenn ein direkter Wert angegeben wird).

PC-relative Lasten

Die andere Möglichkeit, beliebige 32-Bit-Werte in Register zu laden, besteht darin, den Wert direkt im Flash-Speicher abzulegen und von dort mit "ldr" zu laden:

```
@ Ein bisschen Code…
mov r0,… Adresse von Literal ...
ldr r1, [r0]
@ Mehr Code…
Wörtlich:
    .word 0x12345678
```

Es gibt jedoch ein Hühnchen-und-Ei-Problem - die Adresse von "Literal" ist eine 32-Bit-Zahl selbst, also wie lade ich sie in r0? Zum Glück gibt es ein Register, das eine Zahl enthält, die der benötigten nahe kommt - der Programmzähler (PC, r15) zeigt die Adresse des Befehls an, der gerade ausgeführt wird. Durch Lesen und Hinzufügen eines kleinen Versatzes, der in den Befehl selbst passt, kann die Adresse von "Literal" erhalten werden, vorausgesetzt, dass "Literal" nahe genug liegt. Betrachten Sie das folgende Beispiel für die EnableClockGPIOA-Funktion:

```
.align 2
```

```
.type EnableClockGPIOA,% Funktion
EnableClockGPIOA:
füge r1, pc, # 12 hinzu
ldr r1, [r1]
ldr r0, [r1]
orr r0, r0, # RCC_APB2ENR_IOPAEN
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren
bx lr @ Zurück zum Anrufer

.align 2
.word RCC_APB2ENR
```

Der 32-Bit-Wert „RCC_APB2ENR“ wird im Flash-Speicher gespeichert. Der "add" -Befehl wird verwendet, um den Offset 12 zur Adresse des Befehls selbst zu addieren, um die Adresse des 32-Bit-Werts zu erhalten, der dann über "ldr" geladen wird. Der Versatz 12 ist tatsächlich nicht einfach zu berechnen und hängt sogar von der Ausrichtung des Befehls "add" selbst ab (daher ".align", um ein konsistentes Beispiel zu gewährleisten). Der Assembler ist in der Lage, die Berechnung selbst durchzuführen, wofür die Anweisung "adr" verwendet wird:

```
.align 2
.type EnableClockGPIOA,% Funktion
EnableClockGPIOA:
adr r1, LiteralRCC_APB2ENR
ldr r1, [r1]
ldr r0, [r1]
orr r0, r0, # RCC_APB2ENR_IOPAEN
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren
bx lr @ Zurück zum Anrufer

.align 2
LiteralRCC_APB2ENR:
.word RCC_APB2ENR
```

Die Bezeichnung LiteralRCC_APB2ENR bezieht sich auf die Adresse des 32-Bit-Wertes im Speicher. "Adr" ist eigentlich eine Variante von "add", die den Assembler anweist, den Offset zu berechnen und in die Anweisung selbst einzufügen. Der Prozessor fügt ihn dann zum PC hinzu und schreibt das Ergebnis in r1. Diese Adresse wird dann von "ldr" verwendet.

Der Befehl "adr" ist nützlich, wenn die Adresse eines Literal explizit benötigt wird. Im Blinker-Programm kann es beispielsweise verwendet werden, um die Adressen des Arrays abzurufen:

```
adr r7, BlinkTable @ Adresse von "BlinkTable" in r7 verschieben
adr r8, BlinkTableEnd @ Adresse von "BlinkTableEnd" in r8 verschieben
```

Zum Laden eines einzelnen Wertes wird die Adresse jedoch nicht benötigt. In diesem Fall können "adr" und "ldr" kombiniert werden:

```
.type EnableClockGPIOA,% Funktion
EnableClockGPIOA:
ldr r1, LiteralRCC_APB2ENR
ldr r0, [r1]
orr r0, r0, # RCC_APB2ENR_IOPAEN
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren
bx lr @ Zurück zum Anrufer

.align 2
LiteralRCC_APB2ENR:
.word RCC_APB2ENR
```

Diese spezielle Variante von "ldr" ermöglicht es dem Assembler, den Offset wie bei "adr" zu berechnen, ihn zur Laufzeit zu "PC" hinzuzufügen und die an der Adresse gefundenen Daten in r1 zu laden. Dies ist viel einfacher als die erste Variante, da alle Berechnungen automatisch durchgeführt werden. Es ist immer noch etwas umständlich, drei Zeilen zu schreiben, um einen einzelnen 32-Bit-Wert zu erhalten. Daher bietet der Assembler diese bereits eingeführte Syntax an:

```
ldr r1, = RCC_APB2ENR
```

Dies ist ein spezieller Befehl für den Assembler. Nach Möglichkeit lädt der Assembler den Wert mit der Anweisung „mov“ oder „mvn“. Wenn der Wert nicht passt, wird er in den Flash-Speicher geschrieben, und es

wird eine Anweisung "ldr" wie oben verwendet. In diesem Fall entspricht die Syntax "ldr rX, = ..." der Kombination aus der Angabe eines Labels für den Wert, der Direktive ".word" und "ldr rX, <Label>". Daher ist diese Syntax normalerweise der beste Weg, um Sofort zu laden.

Der Assembler platziert die Literale am Ende der Datei. Wenn die Datei lang ist, ist der Offset zu lang für die Anweisungen "ldr" und "adr", und der Assembler gibt einen Fehler aus. Sie können den Assembler anweisen, alle bisher deklarierten Literale mithilfe der Anweisung ".ltorg" an einer bestimmten Stelle zu platzieren. Es wird empfohlen, nach jeder Funktion ein ".ltorg" (nach dem "bx lr") einzufügen - stellen Sie nur sicher, dass die Ausführung dort niemals ankommt. Wenn eine einzelne Funktion so lang ist, dass ein ".ltorg" am Ende zu weit vom "ldr" / "adr" am Anfang entfernt ist, können Sie irgendwo in der Mitte ein ".ltorg" platzieren und mit " b ”.

Zusammenfassend können die folgenden Regeln dazu beitragen, das Laden von Literalen effizienter zu gestalten

- Vermeiden Sie wörtliche Belastungen, wenn möglich; versuchen Sie, die benötigten Werte aus anderen bereits geladenen Werten zu berechnen, möglicherweise mithilfe der Offset-Adressierung in "ldr" / "str"
- Wenn Sie auf mehrere Register eines einzelnen Peripheriemoduls zugreifen, laden Sie dessen Basisadresse einmal und verwenden Sie die Offset-Adressierung, um auf die einzelnen Register zuzugreifen
- Wenn Sie einen Zeiger auf eine Stelle im Flash-Speicher benötigen, versuchen Sie es mit „adr“.
- Wenn Geschwindigkeit wichtig ist, laden Sie den Wert mit "movw" + "movt"
- Verwenden Sie andernfalls "ldr rX, = ...", damit der Assembler die optimale Codierung auswählt
- Fügen Sie nach jeder Funktion ".ltorg" ein

Der Befehl „ldr... =“ kann auch zum Laden eines beliebigen 32-Bit-Sofortwerts in den PC verwendet werden, um einen Sprung zu dieser Adresse zu bewirken, indem einfach „pc“ als Zielregister angegeben wird. Wenn Sie eine gewöhnliche Verzweigung (über "b" oder "bl") zu einer Funktion ausführen, deren Adresse zu weit von der aktuellen Codestelle entfernt ist, fügt der Linker eine "Wrapper" -Funktion ein, die genau das tut, um die "ferne" Funktion auszuführen. springen. Diese Funktion wird als "Furnier" bezeichnet.

Der SysTick-Timer

Ein wichtiger Aspekt vieler eingebetteter Systeme ist die Steuerung des Timings technischer Prozesse. In dem Blinker-Beispiel wurde die Zeitsteuerung der LED-Blitze gehandhabt, indem der Prozessor Dummy-Anweisungen ausführte, um die Zeit abzulaufen. Es ist jedoch praktisch unmöglich, die Laufzeit eines Codeteils auf einem komplexen Prozessor wie ARM genau vorherzusagen, und die Laufzeit kann zwischen mehreren Läufen variieren und hängt vom tatsächlichen Mikrocontroller und seiner Konfiguration ab. Für einen einfachen LED-Blinker kann dies akzeptabel sein, jedoch nicht für z. ein Closed-Loop-Controller für einen mechanischen Schauspieler. Daher verfügen fast alle Mikrocontroller und auch Anwendungsprozessoren über einen oder mehrere Hardware-Timer, mit denen die Zeit unabhängig von der Ausführungsgeschwindigkeit der Software gemessen werden kann. Die Timer-Funktionen variieren stark zwischen den verschiedenen Prozessoren. Diese Grundidee besteht jedoch darin, einen digitalen Zähler bei jedem Taktzyklus zu erhöhen oder zu verringern und ein Ereignis auszulösen, wenn er einen bestimmten Wert erreicht.

Alle ARMv7-M-Prozessoren verfügen über den sogenannten „SysTick“ -Timer als Bestandteil des Prozessorkerns. Dies ist ein ziemlich einfacher 24-Bit-Timer, der von einem konfigurierbaren Wert zurück auf Null zählt, dann auf diesen Wert zurücksetzt und ein Ereignis auslöst. Dieser Timer wird häufig als Zeitbasis für RTOS oder andere Laufzeitbibliotheken verwendet. Der Timer verwendet drei Peripherieregister: „RVR“ enthält den Wert, ab dem heruntergezählt werden soll. "CVR" enthält den aktuellen Wert und "CSR" enthält einige Status- und Steuerbits. Der Timer kann für die Funktion „Verzögerung“ wie folgt verwendet werden:

```
SCS = 0xe000e000
SCS_SYST_CSR = 0x10
SCS_SYST_RVR = 0x14
SCS_SYST_CVR = 0x18

@ Parameter: r0 = Anzahl der Iterationen
Typ Verzögerung,% Funktion
Verzögern:
ldr r1 = SCS
addiere r0, r0, r0, lsl # 1

str r0, [r1, #SCS_SYST_RVR]
ldr r0, = 0
str r0, [r1, #SCS_SYST_CVR]

ldr r0, = 5
str r0, [r1, #SCS_SYST_CSR]

DelayLoop:
```

```
ldr r0, [r1, #SCS_SYST_CSR]
tst r0, # 0x10000
beq DelayLoop

ldr r0, = 0
str r0, [r1, #SCS_SYST_CSR]

bx lr
```

Der SysTick ist Teil des "System Control Space" (SCS). Die SCS-Basisadresse ist als Symbol definiert, ebenso die relativen Adressen der Register. Der Zählwert wird in „RVR“ gespeichert, danach muss „CVR“ auf Null gesetzt werden. Der Timer wird gestartet, indem "5" in das "CSR" -Register geschrieben wird. Die Schleife liest wiederholt das "CSR" -Register und fährt fort, bis Bit 16 gesetzt ist. Der Befehl "tst" wird verwendet, um eine "and" -Operation mit dem Registerinhalt und einem unmittelbaren Wert durchzuführen, ohne das Ergebnis zu behalten, während nur die Flags aktualisiert werden. Am Ende wird das Register „CSR“ auf Null gesetzt, um den Timer zu deaktivieren. Die Anweisung "add" am Anfang wird verwendet, um den Zählwert mit 3 zu multiplizieren: r0 wird um eins nach links verschoben, d. H. Mit zwei multipliziert und dann wie in r0 * 2 ^ 1 + r0 zu sich selbst addiert. Dies ist ein gängiger Trick, um schnell mit Konstanten zu multiplizieren. Durch die Einbeziehung dieser Multiplikation ist die Dauer dieselbe wie bei der vorherigen "Verzögerungs" -Variante, die auf diesem Mikrocontroller ungefähr 3 Zyklen pro Schleifeniteration verwendet.

Das Timing auf diese Weise zu verwalten (oder eine andere Art von „Verzögerung“) ist immer noch nicht sehr genau. Die Zeit, die benötigt wird, um die Funktion aufzurufen, den Timer zu starten, zurückzukehren und die Stifte zu setzen, wird zur tatsächlichen Dauer addiert und kann auch jedes Mal variieren. Die Zeitfehler häufen sich mit der Zeit - eine so implementierte Uhr geht schnell schief. Der richtige Weg, um ein genaues Timing zu erreichen, besteht darin, den Timer einmal zu starten, ihn kontinuierlich laufen zu lassen und auf seine Ereignisse zu reagieren. Die vom Mikrocontroller verwendete interne Taktquelle ist auch ziemlich ungenau (bis zu 2,5% Abweichung), was durch einen Quarzkristall (typische Genauigkeit von beispielsweise 0,005%) verbessert werden kann, der später behandelt wird. Um auf Ereignisse zu reagieren, anstatt eine Funktion aufzurufen, die Dummy-Code ausführt, muss der Programmcode umstrukturiert werden, ohne dass eine „Verzögerungsfunktion“ verwendet wird.

Dazu wird der Timer beim Programmstart einmalig gestartet und weiter ausgeführt. Warten Sie nach dem Setzen des LED-Pins auf das Timer-Ereignis und wiederholen Sie den Vorgang. Im letzten Beispiel werden die Werte 3000000 und 7500000 für das Timer-Register verwendet (3x1000000 bzw. 3x2500000). Das Ändern des Timer-Werts bei kontinuierlichem Betrieb ist problematisch, daher sollte ein fester Wert verwendet werden. Um eine variable Blinkerdauer zu erreichen, müssen mehrere Timerereignisse gezählt werden. Der größte gemeinsame Nenner der beiden Zahlen ist 1500000. Um die beiden unterschiedlichen Zeiten zu erreichen, müssen 2 bzw. 5 Timerereignisse registriert werden. Da diese Nummern in ein einzelnes Byte passen, werden die Tabelleneinträge und die entsprechenden Zugriffsanweisungen in Byte geändert. Eine Funktion "StartSysTick" ist implementiert, um den Timer einmal zu starten, und eine Funktion "WaitSysTick", um auf eine bestimmte Anzahl von Timer-Ereignissen zu warten:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

RCC_APB2ENR = 0x40021018
RCC_APB2ENR_IOPAEN = 4
GPIOA_CRH = 0x40010804

GPIOA_BSRR = 0x40010810
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000


GPIOx_CRx_GP_PP_2MHz = 2

SCS = 0xe000e000
SCS_SYST_CSR = 0x10
SCS_SYST_RVR = 0x14
SCS_SYST_CVR = 0x18
TimerValue = 1500000

.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:
bl EnableClockGPIOA
bl PA8 konfigurieren
ldr r0, = TimerValue
```

```
bl StartSysTick
bl Blink
b.

Typ Blinken,% Funktion
Blinken:
drücke {r4-r8, lr}
ldr r4, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
ldr r5, = GPIOx_BSRR_BS8 @ Wert registrieren, um Pin auf High zu setzen
ldr r6, = GPIOx_BSRR_BR8 @ Wert registrieren, um Pin auf Low zu setzen
adr r7, BlinkTable @ Adresse von "BlinkTable" in r8 verschieben
adr r8, BlinkTableEnd @ Adresse von "BlinkTableEnd" in r9 verschieben


BlinkLoop:
str r5, [r4] @ Setzen Sie BS8 in GPIOA_BSRR auf 1, um PA8 hoch zu setzen

ldrb r0, [r7], # 1 @ Iterationen der Ladeverzögerung von Tabelle und Inkrementadresse
bl WaitSysTick

str r6, [r4] @ Setzen Sie BR8 in GPIOA_BSRR auf 1, um PA8 auf Low zu setzen

ldrb r0, [r7], # 1 @ Iterationen der Ladeverzögerung von Tabelle und Inkrementadresse
bl WaitSysTick

cmp r7, r8
blo BlinkLoop

pop {r4-r8, pc}

.align 2
.type BlinkTable,% Objekt
BlinkTable:
Byte 2, 2, 2, 2, 2, 2
.byte 5, 2, 5, 2, 5, 2
Byte 2, 2, 2, 2, 2, 2
BlinkTableEnd:
.align 2

.type EnableClockGPIOA,% Funktion
EnableClockGPIOA:
ldr r1, = RCC_APB2ENR
ldr r0, [r1]
orr r0, r0, # RCC_APB2ENR_IOPAEN
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren
bx lr @ Zurück zum Anrufer

.type ConfigurePA8,% Funktion
ConfigurePA8:
ldr r1, = GPIOA_CRH
ldr r0, [r1]
und r0, # 0xfffffff0
orr r0, # GPIOx_CRx_GP_PP_2MHz
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2
bx lr
.ltorg

@ r0 = Countdown-Wert für den Timer
.type InitializeSysTick,% -Funktion
StartSysTick:
ldr r1 = SCS

str r0, [r1, #SCS_SYST_RVR]
ldr r0, = 0
str r0, [r1, #SCS_SYST_CVR]

ldr r0, = 5
str r0, [r1, #SCS_SYST_CSR]

bx lr

@ r0 = Anzahl der Timer-Ereignisse, auf die gewartet werden soll
.type WaitSysTick,% Funktion
WaitSysTick:
```

```
ldr r1 = SCS

WaitSysTickLoop:
ldr r2, [r1, #SCS_SYST_CSR]
tst r2, # 0x10000
beq WaitSysTickLoop

subs r0, # 1
bne WaitSysTickLoop

bx lr
```

Beispielname: „BlinkSysTick“

Auf diese Weise wird die Blinkerfrequenz mit der gegebenen Taktquelle so stabil und genau wie möglich sein.

Ausnahmen & Interrupts

Ausnahmen und Interrupts spielen eine wichtige Rolle in der Low-Level-Entwicklung. Sie bieten der Hardware die Möglichkeit, Ereignisse wie empfangene Datenblöcke oder ein Timer-Ereignis an die Software zu melden. In ARM sind Interrupts eine Untergruppe von Ausnahmen - es gibt einige Ausnahmen auf Systemebene, die sich hauptsächlich mit Prozessorfehlern und der Bereitstellung von Betriebssystemunterstützung befassen, während Interrupts „spezielle“ Ausnahmen für Ereignisse sind, die von Peripheriemodulen signalisiert werden. Wenn Sie „normale“ Mikrocontroller-Software schreiben, arbeiten Sie meist mit Interrupts.

Ausnahmen (und Unterbrechungen) unterbrechen den normalen Programmfluss und veranlassen den Prozessor, einen anderen Teil des Codes auszuführen, der als Ausnahmebehandlungsroutine oder Interrupt Service Routine (ISR) bezeichnet wird (auch für Ausnahmen auf Systemebene, die keine Unterbrechungen sind). Nach der Behandlung des angegebenen Ereignisses kehrt der ISR normalerweise zurück und der normale Programmfluss wird fortgesetzt. Da Ausnahmen das Programm jederzeit unterbrechen können, befinden sich Daten (und Peripheriegeräte) möglicherweise in einem inkonsistenten Zustand. Daher muss besonders darauf geachtet werden, dass der Programmstatus in einem ISR nicht beschädigt wird. Der ARMv7-M-Prozessor (einschließlich des Cortex-M3) bietet eine ausgereifte Unterstützung für Ausnahmen mit konfigurierbaren Prioritäten und verschachtelten Ausnahmeaufrufen. In diesem Kapitel werden nur die Grundlagen für die Verwendung von Ausnahmen behandelt.

In ARMv7-M werden Ausnahmebehandlungsroutinen als reguläre Funktionen implementiert. Beispiel:

```
.type SysTick_Handler,% Funktion
.global SysTick_Handler
SysTick_Handler:
@ Event bearbeiten ...
bx lr
```

Wie jede andere Funktion hat sie eine Bezeichnung, kehrt mit "bx lr" zurück und wird mit ".global" auch für andere Quelldateien global sichtbar gemacht. Die Funktion „.type...%“ wird hier aus dem gleichen Grund benötigt wie beim bereits erwähnten „Reset_Handler“. Ausnahmebehandlungsroutinen können sich neben anderen regulären Funktionen an einer beliebigen Stelle im Flash-Speicher befinden. Um dem Prozessor mitzuteilen, wo sich die Ausnahmebehandlungsroutinen für die verschiedenen Ausnahmetypen befinden, muss die Vektortabelle angepasst werden. Bisher war die Vektortabelle definiert als:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.section .VectorTable, "a"
.word _StackEnd
.word Reset_Handler
.space 0xe4
```

Denken Sie daran, dass das erste 32-Bit-Wort im Flash-Speicher den anfänglichen Stapelzeiger enthält (definiert über ".word _StackEnd") und das zweite Wort die Adresse der ersten Anweisung des Programms enthält (definiert über ".word Reset_Handler"). Eigentlich ist das Zurücksetzen der Steuerung auch eine Ausnahme, und der Code, der nach dem Zurücksetzen (oder Starten) ausgeführt wird, ist der Handler für die Zurücksetzungsausnahme (daher der Name „Reset_Handler“). Die nächsten 228 Bytes Flash-Speicher enthalten 57 32-Bit-Adressen der Handler der anderen Ausnahmen, einschließlich Interrupts. Die Direktive ".space" füllt diese nur mit Nullen. Um dem Prozessor die Adresse eines Ausnahmebehandlers mitzuteilen, muss der entsprechende Eintrag in dieser Tabelle auf diese Adresse gesetzt werden. In Kapitel 10.1.2, Tabelle 63 des Controller-Referenzhandbuchs wird das Format der Vektortabelle und die Adresse der Ausnahmebedingung

definiert. Im STM32F103RB / C8 sind nur die Interrupts bis zur Position 42 vorhanden, wie in Kapitel 2.3.5 des Datenblattes definiert. Alles von „TIM8_BRK“ ist nur bei größeren Steuerungen vorhanden. Gemäß der Tabelle muss die Adresse des SysTick-Ausnahmehandlers an der Position 0x3C relativ zum Beginn des Flash-Speichers platziert werden. Da die ersten 8 Bytes bereits belegt sind, werden nach diesen ersten 8 Bytes 0x34 Bytes Speicherplatz benötigt.

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.section .VectorTable, "a"
.word _StackEnd
.word Reset_Handler
.space 0x34
.word SysTick_Handler
.space 0xac
```

Mit dieser Änderung wird die SysTick_Handler-Funktion jetzt als Handler für die SysTick-Ausnahme deklariert. Standardmäßig löst der SysTick-Timer keine Ausnahme aus. Dazu müssen Sie im Register SCS_SYST_CSR das Bit 2 setzen. Wenn Sie die Logik für den Blinker in den ISR des Timers einfügen, erhalten Sie einen Interrupt-basierten Blinker:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

RCC_APB2ENR = 0x40021018
RCC_APB2ENR_IOPAEN = 4
GPIOA_CRH = 0x40010804

GPIOA_BSRR = 0x40010810
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000


GPIOx_CRx_GP_PP_2MHz = 2

SCS = 0xe000e000
SCS_SYST_CSR = 0x10
SCS_SYST_RVR = 0x14
SCS_SYST_CVR = 0x18
TimerValue = 1500000

.Daten
Variablen:
BlinkStep:
.space 1
TimerEvents:
.space 1

.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:
ldr r0, = Variablen
ldr r1 = 0
str r1, [r0, # (BlinkStep-Variablen)]
ldr r1, BlinkTable
str r1, [r0, # (TimerEvents-Variablen)]

bl EnableClockGPIOA
bl PA8 konfigurieren

ldr r1, = GPIOx_BSRR_BS8
ldr r0, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
str r1, [r0]


ldr r0, = TimerValue
bl StartSysTick
SleepLoop:
wfi
```

```
b SleepLoop


.type SysTick_Handler,% Funktion
.global SysTick_Handler
SysTick_Handler:
ldr r0, = SCS
ldr r0, [r0, #SCS_SYST_CSR]
tst r0, # 0x10000
beq Return

ldr r0, = Variablen

ldrb r1, [r0, # (BlinkStep-Variablen)]

cmp r1, # (BlinkTableEnd-BlinkTable)
bhs Rückkehr

ldrb r3, [r0, # (TimerEvents-Variablen)]
subs r3, # 1

es ist ne
strbne r3, [r0, # (TimerEvents-Variablen)]
bne Rückkehr

füge r1, # 1 hinzu
cmp r1, # (BlinkTableEnd-BlinkTable)
bhs SkipRestart

ldr r2, = BlinkTable
ldrb r3, [r2, r1]
strb r3, [r0, # (TimerEvents-Variablen)]

SkipRestart:
strb r1, [r0, # (BlinkStep-Variablen)]

unds r1, # 1
ite Gl
ldreq r1, = GPIOx_BSRR_BS8
ldrne r1, = GPIOx_BSRR_BR8

ldr r0, = GPIOA_BSRR @ Adresse von GPIOA_BSRR laden
str r1, [r0]

Rückkehr:
bx lr

.align 2
BlinkTable:
Byte 2, 2, 2, 2, 2, 2
.byte 5, 2, 5, 2, 5, 2
Byte 2, 2, 2, 2, 2
BlinkTableEnd:

.align 2

.type EnableClockGPIOA,% Funktion
EnableClockGPIOA:
ldr r1, = RCC_APB2ENR
ldr r0, [r1]
orr r0, r0, # RCC_APB2ENR_IOPAEN
str r0, [r1] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren
bx lr @ Zurück zum Anrufer

.type ConfigurePA8,% Funktion
ConfigurePA8:
ldr r1, = GPIOA_CRH
ldr r0, [r1]
und r0, # 0xfffffff0
orr r0, # GPIOx_CRx_GP_PP_2MHz
str r0, [r1] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2
bx lr
.ltorg
```

```
@ r0 = Countdown-Wert für den Timer
.type InitializeSysTick,% -Funktion
StartSysTick:
ldr r1 = SCS

str r0, [r1, #SCS_SYST_RVR]
ldr r0, = 0
str r0, [r1, #SCS_SYST_CVR]

ldr r0, = 7
str r0, [r1, #SCS_SYST_CSR]

bx lr
```

Beispielname: "BlinkSysTickInterrupt"

Der reguläre Programmablauf besteht nun nur noch aus der Initialisierung der Peripherie, des Timers und des ersten Schritts des Blinkers (Setzen des Pins auf High). Danach sollte der Prozessor nur noch auf Ausnahmen warten, was durch eine einfache Endlosschleife erreicht wird. Der Befehl "wfi" hält den Prozessor an. Wenn eine Ausnahme auftritt, wird der Prozessor aufgeweckt, das ISR ausgeführt und die Ausführung nach dem "wfi" zurückgegeben. Daher wird „wfi“ normalerweise wie gezeigt in eine Endlosschleife gelegt. Diese Technik kann den Stromverbrauch des Prozessors erheblich senken, da er nur ausgeführt wird, wenn etwas getan werden muss, wie dies durch Interrupts angezeigt wird. Das ISR prüft zuerst, ob das Interrupt-Flag im Timer-Register gesetzt ist - dies ist notwendig, da Ausnahmen manchmal "unechte" auftreten können, d. H. Ohne dass ein tatsächliches Ereignis dies verursacht. Die Entscheidung, ob der Pin-Status gesetzt oder zurückgesetzt werden soll, wird anhand des niedrigsten Bits des Tabellenindex getroffen, sodass der Ausgang zwischen 1 und 0 wechselt.

Der Code im ISR muss wissen, welcher Schritt in der Blinksequenz gerade aktiv ist und wie viele Timer-Ereignisse bereits im aktuellen Schritt aufgetreten sind. Daher werden zwei 1-Byte-Variablen im RAM gespeichert. Um darauf zuzugreifen, wird die Offset-Adressierung verwendet, wobei r0 die Basisadresse der Variablen im Speicher enthält und die Offsets in "ldrb" und "strb" entsprechend eingestellt werden. Die letzte Nummer der Blinksequenztabelle entfällt, da sie eigentlich überflüssig ist, da nach Ablauf der letzten Verzögerung keine Aktion ausgeführt wird. Da die Tabellengröße jetzt ungerade ist, ist eine Direktive ".align" erforderlich. Es ist auf jeden Fall eine gute Idee, nach der Datenausgabe immer „.align“ zu setzen.

Da zu jedem Zeitpunkt des regulären Programmablaufs Ausnahmen auftreten können, enthalten die Prozessorregister möglicherweise einige Daten, die nach der Rückkehr des Ausnahmebehandlungsprogramms verwendet werden. Wenn der Ausnahmebehandler also etwas in die Register schreibt, müssen diese bei der Rückkehr von der Ausnahme wiederhergestellt werden. Beim Eintritt in eine Ausnahme speichern die Cortex-M3 / 4-Prozessoren automatisch die Register r0-r3, r12, r14 (LR) und APSR (einschließlich der Flags) auf dem Stapel. Das Verbindungsregister ist mit einem speziellen "Dummy" -Wert gefüllt, und wenn der Ausnahmebehandler mit diesem Wert über "bx lr" zurückkehrt, stellt der Prozessor den vorherigen Zustand der Register wieder her. Dies bedeutet effektiv, dass Sie Ausnahmebehandlungsroutinen wie jede andere Funktion implementieren können, d. H. Sie können r0-r3, r12 und die Flags frei überschreiben und r4-r11 und LR bei Bedarf drücken / einfügen.

Makros

Der Assembler bietet einige Mechanismen, um die Entwicklung von Assemblersprachen zu vereinfachen. Eines davon sind Makros, mit denen Sie Assembler-Code-Snippets definieren können, die Sie bei Bedarf problemlos einfügen können. Der Code im Makro ähnelt zwar Funktionsaufrufen, wird jedoch bei jeder Verwendung des Makros kopiert. Verwenden Sie sie daher nicht zu häufig. Makros werden mit ".macro" gestartet und enden mit der nächsten ".endm" -Direktive. Das folgende Makro setzt beispielsweise den LED-Pin auf 0 oder 1:

```
.macro SETLED-Wert
ldr r0, = GPIOA_BSRR
ldr r1, = ((( \ value) << 24) | (\ value << 8))
str r1, [r0]
.endm


SETLED 0
SETLED 1
.endm
```

Der Makroname wird als "SETLED" definiert, und ein einzelner Parameter mit dem Namen "value" wird angegeben. Durch Eingabe von "\ type" wird der Wert des Parameters im Makrotext ersetzt. Eine gewisse Bitverschiebung wird verwendet, um das richtige Bitmuster zum Schreiben in BSRR zu berechnen, um den Pin entsprechend zu setzen oder zurückzusetzen.

Schwache Symbole

Wie bereits erläutert, werden in Assemblydateien definierte Beschriftungen in den Objektcodedateien in Symbole übersetzt, die vom Linker aufgelöst werden. Manchmal ist es wünschenswert, eine "Standard" - oder "Fallback" -Implementierung einer Funktion (oder eines Datenblocks) bereitzustellen, die nur verwendet wird, wenn keine andere Implementierung angegeben ist. Dies kann erreicht werden, indem die „Fallback“ -Variante mit „.weak“ markiert wird:

```
Typ Funktion1,% Funktion
.globale Funktion1
.schwache Funktion1
Funktion1:
@ Standardimplementierung…
…
bl Funktion1 @ Rufen Sie die Funktion auf
```

Nur mit diesem Code wird „Function1“ normal verwendet. Wenn Sie eine andere Funktion mit demselben Namen in eine andere Assembly-Quelldatei einfügen, wird diese zweite Variante verwendet.

Symbol-Aliase

Es ist auch möglich, Aliase für Symbole mit ".thumb_set" zu definieren, wodurch die Adresse entsprechend festgelegt wird. Zum Beispiel:

```
Typ Funktion1,% Funktion
.globale Funktion1
Funktion1:
@ Ein bisschen Code

.thumb_set Function2, Function1
…
bl Funktion2 @ Rufen Sie die Funktion auf
```

Beim Versuch, "Function2" aufzurufen, gibt der Linker automatisch die Adresse von "Function1" ein. Dies kann auch mit „.weak“ kombiniert werden, um einen schwachen Alias zu definieren:

```
Typ Funktion1,% Funktion
.globale Funktion1
Funktion1:
@ Ein bisschen Code

.schwache Funktion2
.thumb_set Function2, Function1
…
bl Funktion2 @ Rufen Sie die Funktion auf
```

Wenn Sie jetzt eine andere „Function2“ in einer anderen Assembly-Quelldatei definieren, wird diese verwendet. Andernfalls wird "Function1" aufgerufen, das Ziel der Aliasdefinition. Dies ist nützlich, wenn Sie eine Standardimplementierung für mehrere verschiedene Funktionen definieren möchten, für die Sie jeweils eine Anweisung ".weak" und eine Anweisung ".thumb_set" benötigen.

Verbesserte Vektortabelle

Die Techniken aus den letzten drei Abschnitten können verwendet werden, um die Definition der Vektortabelle zu verbessern. Die Art und Weise, wie es zuvor definiert wurde, ist nicht sehr flexibel. Um neue Einträge einzufügen, müssen Sie die neuen Lückengrößen und Offsets berechnen. Definieren Sie zunächst einen Standardhandler-ISR, der von Ausnahmen aufgerufen wird, für die kein anderer ISR definiert ist, und ein Makro, das einen Alias für eine Ausnahme mit dem Standardhandler als Ziel definiert, und schließlich mithilfe des Makros eine Tabelle aller Ausnahmen:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.macro defisr name
.global \ name
.schwacher \ name
```

```
.thumb_set \ name, Default_Handler
.word \ name
.endm

.global VectorTable
.section .VectorTable, "a"
.type VectorTable,% Objekt
Vektortabelle:
.word _StackEnd
defisr Reset_Handler
defisr NMI_Handler
defisr HardFault_Handler
defisr MemManage_Handler
defisr BusFault_Handler
defisr UsageFault_Handler
.word 0
.word 0
.word 0
.word 0
defisr SVC_Handler
defisr DebugMon_Handler
.word 0
defisr PendSV_Handler
defisr SysTick_Handler
defisr WWDG_IRQHandler
defisr PVD_IRQHandler
defisr TAMPER_IRQHandler
defisr RTC_IRQHandler
defisr FLASH_IRQHandler
defisr RCC_IRQHandler
defisr EXTI0_IRQHandler
defisr EXTI1_IRQHandler
defisr EXTI2_IRQHandler
defisr EXTI3_IRQHandler
defisr EXTI4_IRQHandler
defisr DMA1_Channel1_IRQHandler
defisr DMA1_Channel2_IRQHandler
defisr DMA1_Channel3_IRQHandler
defisr DMA1_Channel4_IRQHandler
defisr DMA1_Channel5_IRQHandler
defisr DMA1_Channel6_IRQHandler
defisr DMA1_Channel7_IRQHandler
defisr ADC1_2_IRQHandler
defisr USB_HP_CAN1_TX_IRQHandler
defisr USB_LP_CAN1_RX0_IRQHandler
defisr CAN1_RX1_IRQHandler
defisr CAN1_SCE_IRQHandler
defisr EXTI9_5_IRQHandler
defisr TIM1_BRK_IRQHandler
defisr TIM1_UP_IRQHandler
defisr TIM1_TRG_COM_IRQHandler
defisr TIM1_CC_IRQHandler
defisr TIM2_IRQHandler
defisr TIM3_IRQHandler
defisr TIM4_IRQHandler
defisr I2C1_EV_IRQHandler
defisr I2C1_ER_IRQHandler
defisr I2C2_EV_IRQHandler
defisr I2C2_ER_IRQHandler
defisr SPI1_IRQHandler
defisr SPI2_IRQHandler
defisr USART1_IRQHandler
defisr USART2_IRQHandler
defisr USART3_IRQHandler
defisr EXTI15_10_IRQHandler
defisr RTCAlarm_IRQHandler
defisr USBWakeUp_IRQHandler

.Text

.type Default_Handler,% -Funktion
.global Default_Handler
Default_Handler:
bkpt
```

```
b.n Default_Handler
```

Die Tabelle enthält einige leere Einträge, die vom Prozessor nicht verwendet werden. Zu Beginn gibt es noch die Definition für den Initial Stack Pointer und den „Reset_Handler“. Wenn Sie Ihr "vectortable.S" durch diesen Code ersetzen, erhalten Sie eine "richtige" Vektortabelle. Der „SysTick_Handler“ funktioniert weiterhin wie bisher, und wenn Sie einen anderen ISR definieren müssen, z. B. für USART1, definieren Sie einfach eine Funktion mit dem genauen Namen „USART1_IRQHandler“. Die Adresse dieser Funktion wird automatisch in die Vektortabelle eingetragen. Wenn eine Ausnahme ohne einen entsprechenden ISR auftritt, wird der "Default_Handler" aufgerufen, der die Anweisung "bkpt" verwendet, um einen Haltepunkt über den angehängten Debugger zu erzwingen. Dies hilft beim Debuggen übersehener Ausnahmen, ohne mehrere einzelne Dummy-Handler-Funktionen zu definieren.

.include

Das Einfügen der Register- und Bitdefinitionen ("RCC_APB2ENR", "RCC_APB2ENR_IOPAEN", ...) in jede Assembly-Quelldatei ist redundant und fehleranfällig. Stattdessen können Sie sie in eine separate Datei (z. B. "stm32f103.inc") einfügen und mit der Direktive ".include" darauf verweisen:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.include "stm32f103.inc"

@ Normaler Code ...
```

Der Assembler liest den Code aus der enthaltenen Datei und gibt vor, dass er anstelle der Zeile „.include“ geschrieben wurde. Dies kann zur Verbesserung der Codestruktur beitragen. Während Sie an der Projektstruktur arbeiten, können Sie auch die Definitionen für die GPIO-Register neu strukturieren, um die Offset-Adressierung zu vereinfachen:

```
GPIOA = 0x40010800

GPIOx_CRH = 0x4
GPIOx_BSRR = 0x10
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000
```

Das nächste Beispiel bezieht diese Änderungen in die Adressierung der Register ein.

Lokale Labels

Es kann mühsam sein, eindeutige Bezeichnungen für alle Sprungziele in Funktionen (z. B. für bedingten Code und Schleifen) zu erfinden. Bei Verwendung eines Disassemblers (siehe unten) wird jedes Etikett als eigene Funktion angezeigt. Daher unterstützt der GNU-Assembler lokale Bezeichnungen. Dies sind Bezeichnungen, deren Name nur aus einer Zahl besteht. Lokale Namen müssen nicht eindeutig sein. mehrere Etiketten, die z.B. "1" kann in einer Datei vorhanden sein. Um zu einem lokalen Etikett zu springen, verwenden Sie die Nummer und fügen Sie ein "f" oder "b" hinzu, um anzugeben, ob vorwärts oder rückwärts gesprungen werden soll. Lokale Labels können mit der Direktive ".global" nicht exportiert werden. Der Interrupt-basierte Blinker kann folgendermaßen mit lokalen Bezeichnungen geändert werden:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.include "stm32f103.inc"

TimerValue = 1500000

.Daten
Variablen:
BlinkStep:
.space 1
TimerEvents:
.space 1

.Text
.type Reset_Handler,% Funktion
```

```
.global Reset_Handler
Reset_Handler:
ldr r0, = Variablen
ldr r1 = 0
str r1, [r0, # (BlinkStep-Variablen)]
ldr r1, BlinkTable
str r1, [r0, # (TimerEvents-Variablen)]

bl EnableClockGPIOA
bl PA8 konfigurieren

ldr r1, = GPIOx_BSRR_BS8
ldr r0, = GPIOA @ Adresse von GPIOA_BSRR laden
str r1, [r0, #GPIOx_BSRR]


ldr r0, = TimerValue
bl StartSysTick
1:
wfi
b 1b


.type SysTick_Handler,% Funktion
.global SysTick_Handler
SysTick_Handler:
ldr r0, = SCS
ldr r0, [r0, #SCS_SYST_CSR]
tst r0, # 0x10000
beq 2f

ldr r0, = Variablen

ldrb r1, [r0, # (BlinkStep-Variablen)]

cmp r1, # (BlinkTableEnd-BlinkTable)
bhs 2f

ldrb r3, [r0, # (TimerEvents-Variablen)]
subs r3, # 1

es ist ne
strbne r3, [r0, # (TimerEvents-Variablen)]
bne 2f

füge r1, # 1 hinzu
cmp r1, # (BlinkTableEnd-BlinkTable)
bhs 1f

ldr r2, = BlinkTable
ldrb r3, [r2, r1]
strb r3, [r0, # (TimerEvents-Variablen)]

1:
strb r1, [r0, # (BlinkStep-Variablen)]

unds r1, # 1
ite Gl
ldreq r1, = GPIOx_BSRR_BS8
ldrne r1, = GPIOx_BSRR_BR8

ldr r0, = GPIOA @ Adresse von GPIOA_BSRR laden
str r1, [r0, #GPIOx_BSRR]

2:
bx lr

.align 2
.type BlinkTable,% Objekt
BlinkTable:
Byte 2, 2, 2, 2, 2, 2
.byte 5, 2, 5, 2, 5, 2
Byte 2, 2, 2, 2, 2
BlinkTableEnd:
```

```
.align 2

.type EnableClockGPIOA,% Funktion
EnableClockGPIOA:
ldr r1, = RCC
ldr r0, [r1, # RCC_APB2ENR]
orr r0, r0, # (1 << RCC_APB2ENR_IOPAEN)
str r0, [r1, # RCC_APB2ENR] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren
bx lr @ Zurück zum Anrufer

.type ConfigurePA8,% Funktion
ConfigurePA8:
ldr r1, = GPIOA
ldr r0, [r1, #GPIOx_CRH]
und r0, # 0xfffffff0
orr r0, # GPIOx_CRx_GP_PP_2MHz
str r0, [r1, #GPIOx_CRH] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2
bx lr
.ltorg

@ r0 = Countdown-Wert für den Timer
.type InitializeSysTick,% -Funktion
StartSysTick:
ldr r1 = SCS

str r0, [r1, #SCS_SYST_RVR]
ldr r0, = 0
str r0, [r1, #SCS_SYST_CVR]

ldr r0, = 7
str r0, [r1, #SCS_SYST_CSR]

bx lr
```

Beispielname: "BlinkLocalLabels"

RAM wird initialisiert

Das Blinker-Programm verwendet 2-Byte-Variablen im Speicher, die beim Start auf einen bestimmten Wert initialisiert werden müssen. Bei großen Programmen mit vielen Variablen wird dies schnell schwierig zu warten und auch ineffizient. Assembler und Linker können dazu beitragen, ein „Bild“ zu erstellen, wie der RAM-Inhalt nach der Initialisierung aussehen soll, und dieses Bild neben den normalen Programmdaten im Flash-Speicher ablegen. Beim Start kann dieses Image einfach 1: 1 in einer Schleife in den RAM kopiert werden. Die meisten Programme enthalten viele Variablen, die mit Null initialisiert werden. Das Platzieren eines (möglicherweise großen) Blocks von Nullen im Flash-Speicher ist daher verschwenderisch. Daher wird eine zusätzliche Schleife verwendet, um alle Nullvariablen auf Null zu initialisieren. Beide Techniken werden auch von C- und C ++ -Compilern verwendet, sodass auch dort die Implementierung des Initialisierungscodes erforderlich ist. Ändern Sie zunächst die Deklaration Ihrer Variablen mit ".byte", ".hword" und ".word" und geben Sie den gewünschten Initialisierungswert ein. Variablen, die mit Null initialisiert werden sollen, werden nach einer .bss-Direktive platziert, um sie in den gleichnamigen Abschnitt zu setzen. Sie erhalten keinen Initialisierungswert, sondern nur reservierten Speicherplatz mithilfe von ".space":

```
.Daten
TimerEvents:
Byte 2

.bss
BlinkStep:
.space 1
```

Aus Sicht des Assemblers landen die Initialisierungsdaten - in diesem Fall nur ein Byte mit dem Wert "2" - direkt im RAM. Bei Mikrocontrollern ist dies jedoch nicht möglich, da der RAM beim Start immer zufällige Daten enthält und nicht automatisch initialisiert wird. Um dies zu erreichen, ändern Sie das Linker-Skript wie folgt:

```
ERINNERUNG {
FLASH: ORIGIN = 0x8000000, LÄNGE = 128 KB
SRAM: ORIGIN = 0x20000000, LENGTH = 20K
}
```

```
ABSCHNITTE {
.VectorTable: {
* (. VectorTable)
}> FLASH

.text: {
*(.Text)
. = AUSRICHTEN (4);
}> FLASH

.stack (NOLOAD): {
. =. + 0x400;
_StackEnd =.;
}> SRAM

.Daten : {
_DataStart =.;
*(.Daten);
. = AUSRICHTEN (4);
_DataEnd =.;
}> SRAM AT> FLASH

_DataLoad = LOADADDR (.data);

.bss: {
_BssStart =.;
* (.bss);
. = AUSRICHTEN (4);
_BssEnd =.;
}> SRAM
}
```

Beispielname: „BlinkInitRAM“

Der Stack wurde in einen eigenen Abschnitt mit dem Attribut "NOLOAD" gestellt, da er nicht initialisiert werden muss. Die Daten werden nun in den Bereich ".data" gestellt. Die Anfangsdaten für diesen Abschnitt werden über das Konstrukt „> SRAM AT> FLASH“ in den Flash-Speicher geschrieben. Die Adressen der Symbole im Abschnitt ".data" sind weiterhin die Adressen im RAM, sodass der Zugriff auf die Symbole aus dem Assembly-Code weiterhin funktioniert. Dem Symbol „_DataStart“ wird der Anfang der initialisierten Daten im RAM und dem Symbol „_DataEnd“ das Ende zugewiesen. Mit der Funktion „LOADADDR“ wird der Anfang der Initialisierungsdaten in Flash abgerufen und „_DataLoad“ zugewiesen. Der Abschnitt ".bss" enthält alle Variablen, die mit Null initialisiert werden sollen, und die Symbole "_BssStart" und "_BssEnd" werden auf ihre Anfangs- bzw. Endadresse gesetzt. Da der Anfang und die Größe des Stapels bereits ein Vielfaches von 4 sind, ist auch der Anfang von „.data“. Die Größe von .data darf jedoch nicht ein Vielfaches von 4 sein. Daher wird ein Befehl ". = ALIGN (4)" direkt vor der Definition von "_DataEnd" eingefügt. Dies fügt 0-3 Dummy-Bytes hinzu, indem der Positionszähler erhöht wird, um sicherzustellen, dass die Adresse ein Vielfaches von 4 ist. Dasselbe wird direkt vor „_BssEnd“ und auch am Ende des Abschnitts „.text“ ausgeführt, um dies sicherzustellen "_BssEnd" und "_DataLoad" sind ebenfalls Vielfache von 4.

Das einzige, was noch übrig ist, ist die eigentliche Initialisierung des RAM. Ändern Sie dazu den „Reset_Handler“ wie folgt:

```
.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:
ldr r0, = _DataStart
ldr r1, = _DataEnd
ldr r2, = _DataLoad

b 2f
1: ldr r3, [r2], # 4
str r3, [r0], # 4
2: cmp r0, r1
blo 1b

ldr r0, = _BssStart
ldr r1, = _BssEnd
ldr r2 = 0

b 2f
1: str r2, [r0], # 4
```

```
2: cmp r0, r1
blo 1b

bl EnableClockGPIOA
bl PA8 konfigurieren

ldr r1, = GPIOx_BSRR_BS8
ldr r0, = GPIOA @ Adresse von GPIOA_BSRR laden
str r1, [r0, #GPIOx_BSRR]


ldr r0, = TimerValue
bl StartSysTick
1:
wfi
b 1b
.ltorg
```

Die explizite Initialisierung der Variablen wurde entfernt. Stattdessen werden die im Linker-Skript definierten Adressen für "_DataStart", "_DataEnd" und "_DataLoad" geladen. Dann lädt eine kurze Schleife wiederholt ein Wort aus dem Flash (d. H. Beginnend mit "_DataLoad") und speichert es im RAM (beginnend mit "_DataStart"). Die Adresszeiger werden nach dem Zugriff durch die Anweisungen "ldr" / "str" inkrementiert. Der Zeiger für die RAM-Position wird mit dem Ende des RAM-Bereichs ("_DataEnd") verglichen, um zu entscheiden, ob zum Anfang der Schleife zurückgesprungen werden soll. Zum Starten der Schleife wird direkt zum Vergleich gesprungen; Dies vermeidet die Notwendigkeit, den Vergleich am Anfang und innerhalb der Schleife durchzuführen. Die zweite Schleife führt die Nullinitialisierung des Bereichs zwischen "_BssStart" und "_BssEnd" durch. es funktioniert ähnlich, aber es müssen keine Daten geladen werden.

Leider kann das gezeigte Programm nicht übersetzt werden - da sich die beiden Variablen nun in zwei verschiedenen Abschnitten befinden (".data" und ".bss"), funktioniert die Offset-Adressierung im "SysTick_Handler" nicht mehr. Daher muss eine direkte Adressierung verwendet werden:

```
.type SysTick_Handler,% Funktion
.global SysTick_Handler
SysTick_Handler:
ldr r0, = SCS
ldr r0, [r0, #SCS_SYST_CSR]
tst r0, # 0x10000
beq 2f

ldr r0, = BlinkStep

ldrb r1, [r0]

cmp r1, # (BlinkTableEnd-BlinkTable)
bhs 2f

ldr r0, = TimerEvents
ldrb r3, [r0]
subs r3, # 1

es ist ne
strbne r3, [r0]
bne 2f

füge r1, # 1 hinzu
cmp r1, # (BlinkTableEnd-BlinkTable)
bhs 1f

ldr r2, = BlinkTable
ldrb r3, [r2, r1]
strb r3, [r0]

1:
ldr r0, = BlinkStep
strb r1, [r0]

unds r1, # 1
ite Gl
ldreq r1, = GPIOx_BSRR_BS8
ldrne r1, = GPIOx_BSRR_BR8
```

```
ldr r0, = GPIOA @ Adresse von GPIOA_BSRR laden
str r1, [r0, #GPIOx_BSRR]

2:
bx lr
```

Peripherie-Interrupts

Interrupts, d. H. Ausnahmen, die von Peripheriemodulen aufgerufen werden, benötigen im Vergleich zu den "Kern" -Ausnahmen, einschließlich des SysTicks, etwas mehr Code. Der Interrupt-Controller des Cortex-M (der NVIC) enthält mehrere Register zur Konfiguration dieser Interrupts. Es ist möglich, die Priorität zu konfigurieren und Interrupts manuell auszulösen. Für die meisten Anwendungen muss jedoch nur der gewünschte Interrupt aktiviert werden. Dies erfolgt über die Register „NVIC_ISER0“ bis „NVIC_ISER15“, die im ARMv7M-Architektur-Referenzhandbuch in Kapitel B3.4.4 dokumentiert sind. Jedes dieser Register enthält 32 Bits, mit denen 32 der Interrupts freigegeben werden können. Der STM32F103RB / C8 verfügt über 43 Interrupts, sodass nur zwei der möglichen 16 Register vorhanden sind. Die Anzahl der Interrupts ist in Kapitel 2.3.5 des Datenblattes der Steuerung angegeben. Um einen Interrupt x zu aktivieren, muss das Bit "x mod 32" im Register NVIC_ISER*y* mit y = x / 32 gesetzt werden. Die Adresse dieses Registers lautet 0xE000E100 + y * 4. Wenn die Nummer eines Interrupts in r0 gegeben ist, macht die folgende Funktion genau das:

```
NVIC_ISER0 = 0xE000E100

@ r0 = IRQ-Nummer
.type EnableIRQ,% Funktion
EnableIRQ:
ldr r1, = NVIC_ISER0

movs r2, # 1
und r3, r0, # 0x1F
lsls r2, r2, r3

lsrs r3, r0, # 5
lsls r3, r3, # 2

str r2, [r1, r3]

bx lr
.ltorg
```

Beispielname: "BlinkTIM1"

Der Befehl "und" berechnet "x mod 32", und die folgende Linksverschiebung ("lsls") berechnet den Wert, bei dem Bit "x mod 32" eins ist und alle anderen null sind. Um die Versatzadresse "y * 4" zu berechnen, d. H. "(X / 32) * 4", wird das Register zuerst um 5 Bits nach rechts und dann um 2 Bits nach links zurück verschoben. Dies ist dasselbe, als würden 3 Bits nach rechts verschoben und die unteren 2 Bits auf Null gesetzt. Zwei Schichtbefehle belegen jedoch tatsächlich weniger Programmspeicherplatz. Schließlich wird der berechnete Wert unter Verwendung einer Versatzadressierung in das Register geschrieben.

Zusätzlich zur Aktivierung des Interrupts im NVIC des Prozessorkerns muss dieser auch im Peripheriemodul aktiviert werden. Viele Peripheriemodule unterstützen mehrere verschiedene Ereignisse, von denen jedes einzeln im Peripherieregister aktiviert werden muss. Abhängig von der Steuerung können diese einem einzelnen Prozessor-Interrupt (und damit einem einzelnen ISR) oder mehreren zugeordnet werden und müssen im NVIC entsprechend konfiguriert werden.

In diesem Beispiel wird der Peripherietimer TIM1 des STM32 anstelle des SysTick-Timers verwendet:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.include "stm32f103.inc"

TimerValue = 1500
TimerPrescaler = 1000

.Daten
TimerEvents:
Byte 2

.bss
```

```
BlinkStep:
.space 1

.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:
ldr r0, = _DataStart
ldr r1, = _DataEnd
ldr r2, = _DataLoad

b 2f
1: ldr r3, [r2], # 4
str r3, [r0], # 4
2: cmp r0, r1
blo 1b

ldr r0, = _BssStart
ldr r1, = _BssEnd
ldr r2 = 0

b 2f
1: str r2, [r0], # 4
2: cmp r0, r1
blo 1b

bl EnableClockGPIOA
bl EnableClockTIM1
bl PA8 konfigurieren

ldr r1, = GPIOx_BSRR_BS8
ldr r0, = GPIOA
str r1, [r0, #GPIOx_BSRR]

ldr r0, = TIM1_UP_IRQn
bl EnableIRQ
bl StartTIM1
1:
wfi
b 1b
.ltorg

.type TIM1_UP_IRQHandler,% Funktion
.global TIM1_UP_IRQHandler
TIM1_UP_IRQHandler:
ldr r0, = TIM1
ldr r2, = (~ (1 << TIMx_SR_UIF))

ldr r1, [r0, #TIMx_SR]
bics r1, r2
beq 2f

str r2, [r0, #TIMx_SR]

ldr r0, = BlinkStep

ldrb r1, [r0]

cmp r1, # (BlinkTableEnd-BlinkTable)
bhs 2f

ldr r0, = TimerEvents
ldrb r3, [r0]
subs r3, # 1

es ist ne
strbne r3, [r0]
bne 2f

füge r1, # 1 hinzu
cmp r1, # (BlinkTableEnd-BlinkTable)
bhs 1f

ldr r2, = BlinkTable
```

```
ldrb r3, [r2, r1]
strb r3, [r0]

1:
ldr r0, = BlinkStep
strb r1, [r0]

unds r1, # 1
ite Gl
ldreq r1, = GPIOx_BSRR_BS8
ldrne r1, = GPIOx_BSRR_BR8

ldr r0, = GPIOA
str r1, [r0, #GPIOx_BSRR]

2:
bx lr

.align 2
.type BlinkTable,% Objekt
BlinkTable:
Byte 2, 2, 2, 2, 2, 2
.byte 5, 2, 5, 2, 5, 2
Byte 2, 2, 2, 2, 2
BlinkTableEnd:

.align 2

.type EnableClockGPIOA,% Funktion
EnableClockGPIOA:
ldr r1, = RCC
ldr r0, [r1, # RCC_APB2ENR]
orr r0, r0, # (1 << RCC_APB2ENR_IOPAEN)
str r0, [r1, # RCC_APB2ENR] @ Setzen Sie das IOPAEN-Bit in RCC_APB2ENR auf 1, um GPIOA zu aktivieren
bx lr @ Zurück zum Anrufer

Typ EnableClockTIM1,% Funktion
EnableClockTIM1:
ldr r1, = RCC
ldr r0, [r1, # RCC_APB2ENR]
orr r0, r0, # (1 << RCC_APB2ENR_TIM1EN)
str r0, [r1, # RCC_APB2ENR] @ Setzen Sie das TIM1EN-Bit in RCC_APB2ENR auf 1, um TIM1 zu aktivieren
bx lr @ Zurück zum Anrufer
.ltorg

.type ConfigurePA8,% Funktion
ConfigurePA8:
ldr r1, = GPIOA
ldr r0, [r1, #GPIOx_CRH]
und r0, # 0xfffffff0
orr r0, # GPIOx_CRx_GP_PP_2MHz
str r0, [r1, #GPIOx_CRH] @ Setzen Sie CNF8: MODE8 in GPIOA_CRH auf 2
bx lr
.ltorg

@ r0 = Countdown-Wert für den Timer
.type InitializeSysTick,% -Funktion
StartTIM1:
ldr r0, = TIM1
ldr r1, = (1 << TIMx_CR1_URS)
str r1, [r0, # TIMx_CR1]

ldr r1, = TimerPrescaler
str r1, [r0, #TIMx_PSC]

ldr r1, = TimerValue
str r1, [r0, #TIMx_ARR]

ldr r1, = (1 << TIMx_DIER_UIE)
str r1, [r0, #TIMx_DIER]

ldr r1, = (1 << TIMx_EGR_UG)
str r1, [r0, #TIMx_EGR]
```

```
dsb

ldr r1, = (1 << TIMx_CR1_CEN)
str r1, [r0, # TIMx_CR1]

bx lr
.ltorg

@ r0 = IRQ-Nummer
.type EnableIRQ,% Funktion
EnableIRQ:
ldr r1, = NVIC_ISER0

movs r2, # 1
und r3, r0, # 0x1F
lsls r2, r2, r3

lsrs r3, r0, # 5
lsls r3, r3, # 2

str r2, [r1, r3]

bx lr
.ltorg
```

Die entsprechende Datei stm32f103.inc mit den hinzugefügten Definitionen für die Timer-Register lautet:

```
GPIOA = 0x40010800

GPIOx_CRH = 0x4
GPIOx_BSRR = 0x10
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000


GPIOx_CRx_GP_PP_2MHz = 2

SCS = 0xe000e000
SCS_SYST_CSR = 0x10
SCS_SYST_RVR = 0x14
SCS_SYST_CVR = 0x18

RCC = 0x40021000
RCC_APB2ENR = 0x18
RCC_APB2ENR_IOPAEN = 2
RCC_APB2ENR_TIM1EN = 11

RCC_CR = 0x0
RCC_CR_PLLRDY = 25
RCC_CR_PLLON = 24
RCC_CR_HSERDY = 17
RCC_CR_HSEON = 16
RCC_CR_HSION = 0

RCC_CFGR = 0x04
RCC_CFGR_PLLMUL = 18
RCC_CFGR_USBPRE = 22
RCC_CFGR_PLLXTPRE = 17
RCC_CFGR_PLLSRC = 16
RCC_CFGR_PPRE2 = 11
RCC_CFGR_PPRE1 = 8
RCC_CFGR_HPRE = 4
RCC_CFGR_SWS = 2
RCC_CFGR_SW = 0

FLASH = 0x40022000
FLASH_ACR = 0
FLASH_ACR_PRFTBE = 4
FLASH_ACR_HLFCYA = 3
FLASH_ACR_LATENCY = 0

TIM1 = 0x40012C00
```

```
TIMx_CR1 = 0
TIMx_CR1_ARPE = 7
TIMx_CR1_URS = 2
TIMx_CR1_CEN = 0


TIMx_DIER = 0xC
TIMx_DIER_UIE = 0

TIMx_SR = 0x10
TIMx_SR_UIF = 0

TIMx_EGR = 0x14
TIMx_EGR_UG = 0

TIMx_PSC = 0x28
TIMx_ARR = 0x2C

TIM1_UP_IRQn = 25

NVIC_ISER0 = 0xE000E100
```

Der Quellcode aktiviert die Uhr des Timers im RCC, bevor er konfiguriert wird. Der Timer unterstützt sowohl einen frei konfigurierbaren Vorteiler zum Teilen der Uhr als auch einen frei konfigurierbaren Maximalwert, die beide durch die StartTIM1-Funktion eingestellt werden. Das TIMx_DIER_UIE-Bit wird gesetzt, um den Interrupt für das sogenannte „Update-Ereignis“ freizugeben, das immer dann ausgelöst wird, wenn der Timer den Maximalwert erreicht. Eine feine Folge von Registerzugriffen ist erforderlich, um den Timer mit der richtigen Konfiguration zu starten, ohne jedoch den Interrupt sofort auszulösen: Um die geänderten Einstellungen sofort zu übernehmen, wird das Bit „TIMx_EGR_UG“ gesetzt, um ein „künstliches“ Aktualisierungsereignis auszulösen. Um zu verhindern, dass dies auch den Interrupt auslöst, wird das Bit „TIMx_CR1_URS“ vorher bzw. nachher gesetzt und gelöscht. Der Timer wird gestartet, indem am Ende das Bit „TIMx_CR1_CEN“ gesetzt wird. Davor wird eine Anweisung "dsb" eingefügt. Diese "Data Synchronization Barrier" wartet, bis alle Schreibzugriffe vollständig verarbeitet wurden - normalerweise arbeitet die Prozessor-Pipeline an mehreren Befehlen gleichzeitig. Da die Timer-Konfiguration wirklich abgeschlossen sein muss, bevor der Timer gestartet wird, ist diese Anweisung erforderlich. Es gibt einige andere Situationen, in denen der Prozessor für die Peripherie zu schnell ist und vorübergehend von einem "dsb" angehalten werden muss. Wenn ein Teil des Peripheriezugriffscodes beim Debuggen schrittweise ausgeführt wird, jedoch nicht bei normaler Ausführung, kann ein gut platzierter "dsb" hilfreich sein.

Der ISR „TIM1_UP_IRQHandler“ wird für den Timer verwendet. Es überprüft das Bit "TIMx_SR_UIF", um zu überprüfen, ob tatsächlich ein Aktualisierungsereignis aufgetreten ist. In diesem Fall wird das Register mit dem Wert 0xFFFFFFFE überschrieben, d. H. Alle Bits außer dem UIF-Bit werden mit "1" geschrieben. Das Schreiben von Einsen hat keine Auswirkung auf die Bits in diesem Register, und das Schreiben einer Null löscht das entsprechende Bit. Daher löscht dieser Schreibzugriff das UIF-Bit, behält jedoch die anderen bei. Diese Interrupt-Flags müssen im ISR immer so schnell wie möglich gelöscht werden, da die Peripherie den Interrupt sonst sofort wieder auslösen kann. Der Rest des ISR bleibt gleich.

Analyse-Tools

Wenn Sie direkt mit Linkerskripten und Assembler-Code auf niedriger Ebene arbeiten, ist es häufig erforderlich, die Übersetzungsausgabe direkt zu überprüfen, da Sie sich nicht darauf verlassen können, dass ein Compiler sie automatisch richtig ausführt und das Programm jedes Mal flasht, um zu prüfen, ob sie funktioniert ist nicht der effizienteste Weg. Dies war in der Tat wichtig, um die Beispielcodes für dieses Lernprogramm zu erstellen. Das Paket "binutils", zu dem Assembler und Linker gehören, bietet einige Tools, die bei der Analyse der Assembler- und Linker-Ausgabe hilfreich sind.

Disassembler

Wie der Name schon sagt, ist ein Disassembler das Gegenteil von einem Assembler - er wandelt binären Maschinencode wieder in eine (mehr oder weniger) lesbare Textdarstellung um. Wenn Sie eine vom Assembler oder Linker generierte ELF-Datei in den Disassembler einspeisen, liest dieser die Header-Informationen, um Daten (dh Konstanten) und Code zu unterscheiden, Symbolnamen (und damit Etiketten) abzurufen und sogar zu erkennen, welche Anweisungen vorhanden waren Wird aus welcher Assembly-Quelldatei generiert, wenn es sich um einen Assembler mit Debug-Informationen handelt (dh das Flag "-g" wurde verwendet). Wenn Sie ein binäres Flash-Image zerlegen, verfügt der Disassembler nicht über alle diese Informationen und erzeugt eine viel weniger lesbare Ausgabe und versucht, Datenbytes als Anweisungen zu decodieren.

Der Disassembler von binutils heißt "objdump". Das Aufrufen auf dem Blinker sieht folgendermaßen aus:

```
$ arm-none-eabi-objdump -d -s prog1.elf
```

```
prog1.elf: Dateiformat elf32-littlearm

Inhalt des Abschnitts .VectorTable:
 8000000 00040020 ed000008 ed010008 ed010008 ... ............
 8000010 ed010008 ed010008 ed010008 00000000 ................
 8000020 00000000 00000000 00000000 ed010008 ................
 8000030 ed010008 00000000 ed010008 49010008 ............ I ...
 8000040 ed010008 ed010008 ed010008 ed010008 ................
 8000050 ed010008 ed010008 ed010008 ed010008 ................
 8000060 ed010008 ed010008 ed010008 ed010008 ................
 8000070 ed010008 ed010008 ed010008 ed010008 ................
 8000080 ed010008 ed010008 ed010008 ed010008 ................
 8000090 ed010008 ed010008 ed010008 ed010008 ................
 80000a0 ed010008 ed010008 ed010008 ed010008 ................
 80000b0 ed010008 ed010008 ed010008 ed010008 ................
 80000c0 ed010008 ed010008 ed010008 ed010008 ................
 80000d0 ed010008 ed010008 ed010008 ed010008 ................
 80000e0 ed010008 ed010008 ed010008 ............
Inhalt des Abschnitts .text:
 80000ec 0f481049 104a03e0 52f8043b 40f8043b .H.I.J..R ..; @ ..;
 80000fc 8842f9d3 0d480e49 4ff00002 01e040f8 .B ... H.IO ..... @.
 800010c 042b8842 fbd300f0 47f800f0 4bf84ff4. +. B .... G ... K.O.
 800011c 80710848 01600848 00f058f8 30bffde7 .q.H.`.H..X.0 ...
 800012c 00040020 04040020 f0010008 04040020 ... ... .......
 800013c 08040020 10080140 60e31600 4ff0e020 ... ... @ `... O ..
 800014c 006910f4 803f1dd0 1a480178 b1f1110f .i ...? ... H.x ....
 800015c 18d21948 0378013b 1cbf0370 12e001f1 ... H.x.; ... p ....
 800016c 0101b1f1 110f02d2 144a535c 03701148 ......... JS \ .p.H
 800017c 017011f0 01010cbf 4ff48071 4ff08071 .p ...... O..qO..q
 800018c 0f480160 70470202 02020202 05020502 .H.`pG ..........
 800019c 05020202 02020200 0a490868 40f00400 ......... I.h @ ...
 80001ac 08607047 08490868 20f00f00 40f00200 .`pG.I.h ... @ ...
 80001bc 08607047 04040020 00040020 92010008 .`pG ... ... ....
 80001cc 10080140 18100240 04080140 4ff0e021 ... @ ... @ ... @ O ..!
 80001dc 48614ff0 00008861 4ff00700 08617047 HaO .... aO .... apG
 80001ec 00befde7 ....
Inhalt des Abschnitts .data:
 20000400 02000000 ....
Inhalt des Abschnitts .ARM.attributes:
 0000 41200000 00616561 62690001 16000000 A ... aeabi ......
 0010 05436f72 7465782d 4d330006 0a074d09 .Cortex-M3 .... M.
 0020 02.
Inhalt des Abschnitts .debug_line:
 0000 98000000 02001e00 00000201 fb0e0d00 ................
 0010 01010101 00000001 00000100 70726f67 ............ prog
 0020 312e5300 00000000 000502ec 00000803 1.S ............
 0030 15012121 22212f2f 21222121 30212f21 .. !! "! //!" !! 0! /!
 0040 222f302f 21232130 21036120 2f2f362f "/0/!#!0!.a // 6 /
 0050 030c2e32 030a2e2f 212f2222 222f2221 ... 2 ... /! / "" / "!
 0060 21222121 222f2f22 21212321 222f212f! "!!" // "!! #!" /! /
 0070 30212303 0d9e2121 2f212421 212f2f21 0! # ... !! /! $ !! //!
 0080 03422035 030c2e03 0d2e0311 2e36030b .B 5 ......... 6 ..
 0090 2e30212f 222f2202 01000101 3b000000 .0! / "/" .....; ...
 00a0 02002400 00000201 fb0e0d00 01010101 .. $ ............
 00b0 00000001 00000100 76656374 6f727461 ........ vectorta
 00c0 626c652e 53000000 00000005 02ec0100 ble.S ...........
 00d0 0803d000 01210201 000101 .....! .....
Inhalt des Abschnitts .debug_info:
 0000 22000000 02000000 00000401 00000000 "...............
 0010 ec000008 ec010008 00000000 08000000 ................
 0020 12000000 01802200 00000200 14000000 ...... ".........
 0030 04019c00 0000ec01 0008f001 00082100 ..............!
 0040 00000800 00001200 00000180 ............
Inhalt des Abschnitts .debug_abbrev:
 0000 01110010 06110112 01030e1b 0e250e13 .............% ..
 0010 05000000 01110010 06110112 01030e1b ................
 0020 0e250e13 05000000.% ......
Inhalt des Abschnitts .debug_aranges:
 0000 1c000000 02000000 00000400 00000000 ................
 0010 ec000008 00010000 00000000 00000000 ................
 0020 1c000000 02002600 00000400 00000000 ...... & .........
 0030 ec010008 04000000 00000000 00000000 ................
Inhalt des Abschnitts .debug_str:
```

```
 0000 70726f67 312e5300 2f746d70 2f746573 prog1.S./tmp/tes
 0010 7400474e 55204153 20322e32 392e3531 t.GNU AS 2.29.51
 0020 00766563 746f7274 61626c65 2e5300 .vectortable.S.

Demontage des Abschnitts .text:

080000ec <Reset_Handler>:
 80000ec: 480f ldr r0, [pc, # 60]; (800012c <Reset_Handler + 0x40>)
 80000ee: 4910 ldr r1, [pc, # 64]; (8000130 <Reset_Handler + 0x44>)
 80000f0: 4a10 ldr r2, [pc, # 64]; (8000134 <Reset_Handler + 0x48>)
 80000f2: e003 b.n 80000fc <Reset_Handler + 0x10>
 80000f4: f852 3b04 ldr.w r3, [r2], # 4
 80000f8: f840 3b04 str.w r3, [r0], # 4
 80000fc: 4288 cmp r0, r1
 80000fe: d3f9 bcc.n 80000f4 <Reset_Handler + 0x8>
 8000100: 480d ldr r0, [pc, # 52]; (8000138 <Reset_Handler + 0x4c>)
 8000102: 490e ldr r1, [pc, # 56]; (800013c <Reset_Handler + 0x50>)
 8000104: f04f 0200 mov.w r2, # 0
 8000108: e001 b.n 800010e <Reset_Handler + 0x22>
 800010a: f840 2b04 str.w r2, [r0], # 4
 800010e: 4288 cmp r0, r1
 8000110: d3fb bcc.n 800010a <Reset_Handler + 0x1e>
 8000112: f000 f847 bl 80001a4 <EnableClockGPIOA>
 8000116: f000 f84b bl 80001b0 <ConfigurePA8>
 800011a: f44f 7180 mov.w r1, # 256; 0x100
 800011e: 4808 ldr r0, [pc, # 32]; (8000140 <Reset_Handler + 0x54>)
 8000120: 6001 str r1, [r0, # 0]
 8000122: 4808 ldr r0, [pc, # 32]; (8000144 <Reset_Handler + 0x58>)
 8000124: f000 f858 bl 80001d8 <StartSysTick>
 8000128: bf30 wfi
 800012a: e7fd b.n 8000128 <Reset_Handler + 0x3c>
 800012c: 20000400 .word 0x20000400
 8000130: 20000404 .word 0x20000404
 8000134: 080001f0 .word 0x080001f0
 8000138: 20000404 .word 0x20000404
 800013c: 20000408 .word 0x20000408
 8000140: 40010810 .word 0x40010810
 8000144: 0016e360 .word 0x0016e360

08000148 <SysTick_Handler>:
 8000148: f04f 20e0 mov.w r0, # 3758153728; 0xe000e000
 800014c: 6900 ldr r0, [r0, # 16]
 800014e: f410 3f80 tst.w r0, # 65536; 0x10000
 8000152: d01d beq.n 8000190 <SysTick_Handler + 0x48>
 8000154: 481a ldrr0, [pc, # 104]; (80001c0 <ConfigurePA8 + 0x10>)
 8000156: 7801 ldrb r1, [r0, # 0]
 8000158: f1b1 0f11 cmp.w r1, # 17
 800015c: d218 bcs.n 8000190 <SysTick_Handler + 0x48>
 800015e: 4819 ldr r0, [pc, # 100]; (80001c4 <ConfigurePA8 + 0x14>)
 8000160: 7803 ldrb r3, [r0, # 0]
 8000162: 3b01 subs r3, # 1
 8000164: bf1c itt ne
 8000166: 7003 strbne r3, [r0, # 0]
 8000168: e012 bne.n 8000190 <SysTick_Handler + 0x48>
 800016a: f101 0101 add.w r1, r1, # 1
 800016e: f1b1 0f11 cmp.w r1, # 17
 8000172: d202 bcs.n 800017a <SysTick_Handler + 0x32>
 8000174: 4a14 ldr r2, [pc, # 80]; (80001c8 <ConfigurePA8 + 0x18>)
 8000176: 5c53 ldrb r3, [r2, r1]
 8000178: 7003 strb r3, [r0, # 0]
 800017a: 4811 ldr r0, [pc, # 68]; (80001c0 <ConfigurePA8 + 0x10>)
 800017c: 7001 strb r1, [r0, # 0]
 800017e: f011 0101 ands.w r1, r1, # 1
 8000182: bf0c ite Gl
 8000184: f44f 7180 moveq.w r1, # 256; 0x100
 8000188: f04f 7180 movne.w r1, # 16777216; 0x1000000
 800018c: 480f ldrr0, [pc, # 60]; (80001cc <ConfigurePA8 + 0x1c>)
 800018e: 6001 str r1, [r0, # 0]
 8000190: 4770 bx lr

08000192 <BlinkTable>:
 8000192: 0202 0202 0202 0205 0205 0205 0202 0202 ................
 80001a2:.
```

```
080001a3 <BlinkTableEnd>:
    ...

080001a4 <EnableClockGPIOA>:
 80001a4: 490a ldr r1, [pc, # 40]; (80001d0 <ConfigurePA8 + 0x20>)
 80001a6: 6808 ldr r0, [r1, # 0]
 80001a8: f040 0004 orr.w r0, r0, # 4
 80001ac: 6008 str r0, [r1, # 0]
 80001ae: 4770 bx lr

080001b0 <ConfigurePA8>:
 80001b0: 4908 ldr r1, [pc, # 32]; (80001d4 <ConfigurePA8 + 0x24>)
 80001b2: 6808 ldr r0, [r1, # 0]
 80001b4: f020 000f bic.w r0, r0, # 15
 80001b8: f040 0002 orr.w r0, r0, # 2
 80001bc: 6008 str r0, [r1, # 0]
 80001be: 4770 bx lr
 80001c0: 20000404 .word 0x20000404
 80001c4: 20000400 .word 0x20000400
 80001c8: 08000192 .word 0x08000192
 80001cc: 40010810 .word 0x40010810
 80001d0: 40021018 .word 0x40021018
 80001d4: 40010804 .word 0x40010804

080001d8 <StartSysTick>:
 80001d8: f04f 21e0 mov.w r1, # 3758153728; 0xe000e000
 80001dc: 6148 str r0, [r1, # 20]
 80001de: f04f 0000 mov.w r0, # 0
 80001e2: 6188 str r0, [r1, # 24]
 80001e4: f04f 0007 mov.w r0, # 7
 80001e8: 6108 str r0, [r1, # 16]
 80001ea: 4770 bx lr

080001ec <Default_Handler>:
 80001ec: be00 bkpt 0x0000
 80001ee: e7fd b.n 80001ec <Default_Handler>
```

Das sind viele Informationen. Das Flag "-d" weist objdump an, Codeabschnitte zu disassemblieren, und das Flag "-s" ermöglicht die Ausgabe von Datenabschnitten. Zuerst wird der Inhalt von ".VectorTable" gedruckt. Jeder Zeile wird die Adresse vorangestellt, unter der sich diese Daten im Speicher befinden. Dann werden die 32-Bit-Datenblöcke aus der Vektortabelle ausgegeben. Der Disassembler druckt die Bytes in der Reihenfolge, in der sie im Speicher erscheinen. Da der Cortex-M3 Little Endian verwendet, ist dies umgekehrt. Beispielsweise bezieht sich das gedruckte "ed000008" tatsächlich auf die Adresse "0x080000ed", die die Adresse des "Reset_Handler" mit dem niedrigsten Bit auf eins, da es sich um eine Daumenfunktion handelt. Die meisten Adressen in der Vektortabelle geben die Adresse des Standardhandlers 0x080001ec wieder, mit Ausnahme der Null-Einträge und des SysTick_Handlers. Der Inhalt des Abschnitts ".text" ist die hexadezimale Darstellung des Maschinencodes und kaum lesbar. Der Abschnitt ".data" enthält eine einzelne "Zwei" - dies ist die "02", die in "TimerEvents" eingetragen ist. Der Inhalt von „.ARM.attributes:“ und den verschiedenen Abschnitten „.debug“ ist nicht sehr interessant, da er nicht auf dem Controller landet und nur von den verschiedenen Analyse-Tools gelesen wird, um eine bessere Ausgabe zu erzielen.

Danach folgt die eigentliche Demontage. Dies ist eine Liste aller Anweisungen im Codeabschnitt. Die Liste ist nach den Symbolen in der Eingabedatei gruppiert. Bei C-Code entspricht jedes Symbol normalerweise einer Funktion, sodass jeder Block in der Disassemblierung eine C-Funktion darstellt. Wenn Sie im Assembler-Code nicht lokale Bezeichnungen in eine Funktion einfügen, wird diese Funktion vom Disassembler in mehrere Blöcke aufgeteilt, wodurch das Lesen erschwert wird - der Hauptgrund für die Verwendung lokaler Bezeichnungen. Jede Anweisung wird in eine Zeile innerhalb der Blöcke übersetzt. Die erste Spalte ist die Adresse, an der dieser Befehl gefunden wird. Die nächste Spalte enthält die hexadezimale Darstellung der 2 oder 4 Bytes, die den Maschinencode dieser Anweisung bilden, d. H. Den tatsächlichen Inhalt des Flash-Speichers. Danach folgt eine vom Disassembler abgeleitete Textdarstellung dieser Anweisung. Wenn der Befehl eine Zahl enthält, gibt der Disassembler manchmal ein Semikolon aus, gefolgt von einer Interpretation dieser Zahl. Wenn der Befehl eine PC-relative Adressierung verwendet, ist diese Interpretation die absolute Adresse. Da viele Anweisungen mehrere Schreibweisen haben, kann es zu Abweichungen zwischen dem ursprünglichen Code und der Demontage kommen. Der Disassembler gibt auch Daten aus, wie z. B. die "BlinkTable" und die Literal-Pools als solche. Die Verwendung der Direktive ".type" ist in diesem Fall hilfreich, damit der Disassembler nicht versucht, die Datenbytes als Code zu interpretieren.

objdump kann auch verwendet werden, um rohe Binärdateien zu disassemblieren, die durch Zurücklesen des Flash-Speichers eines Controllers abgerufen werden können. Verwenden Sie dazu diese Befehlszeile:

```
$ arm-none-eabi-objdump -b binär -m arm -D prog1.bin -Mforce-thumb --adjust-vma = 0x08000000
```

Die Adresse der Binärdatei im Flash-Speicher wird angegeben, damit die gedruckten Anweisungsadressen korrekt sind. Da der Disassembler jedoch Daten und Code nicht unterscheiden kann, ist das Ergebnis nur begrenzt von Nutzen. Wenn Sie eine Binärdatei analysieren müssen, ohne eine ELF-Datei oder den Quellcode zu haben, ist ein ausgefeilterer Disassembler wie IDA Pro hilfreich. Wenn Sie den Code haben und nur den Disassembler benötigen, um potenzielle Probleme mit dem Projekt zu identifizieren (insbesondere das Linker-Skript), ist objdump normalerweise ausreichend.

readelf

Das Programm "readelf" ist ein leistungsstarkes Dienstprogramm, mit dem verschiedene Informationen aus ELF-Dateien gelesen und ausgegeben werden können. Die nützlichste Option ist das Flag "-S", mit dem sich eine Zusammenfassung der Abschnitte in der jeweiligen Datei ausdrucken lässt, z.

```
$ arm-none-eabi-readelf -S prog1.elf
Es gibt 15 Abschnittsüberschriften, beginnend mit Offset 0x11268:

Abschnittsüberschriften:
  [Nr] Name Typ Addr Off Größe ES Flg Lk Inf Al
  [0] NULL 00000000 000000 000000 00 0 0 0
  [1] .VectorTable PROGBITS 08000000 010000 0000ec 00 A 0 0 1
  [2] .text PROGBITS 080000ec 0100ec 000104 00 AX 0 0 4
  [3] .stack NOBITS 20000000 020000 000400 00 WA 0 0 1
  [4] .data PROGBITS 20000400 010400 000004 00 WA 0 0 1
  [5] .bss NOBITS 20000404 010404 000004 00 WA 0 0 1
  [6] .ARM.attributes ARM_ATTRIBUTES 00000000 010404 000021 00 0 0 1
  [7] .debug_line PROGBITS 00000000 010425 0000db 00 0 0 1
  [8] .debug_info PROGBITS 00000000 010500 00004c 00 0 0 1
  [9] .debug_abbrev PROGBITS 00000000 01054c 000028 00 0 0 1
  [10] .debug_aranges PROGBITS 00000000 010578 000040 00 0 0 8
  [11] .debug_str PROGBITS 00000000 0105b8 00002f 01 MS 0 0 1
  [12] SYMTAB 00000000 0105e8 0006a0 10 13 45 4
  [13] .strtab STRTAB 00000000 010c88 000550 00 0 0 1
  [14] .shstrtab STRTAB 00000000 0111d8 000090 00 0 0 1
Schlüssel zu Flags:
  W (Schreiben), A (Zuweisen), X (Ausführen), M (Zusammenführen), S (Zeichenfolgen), I (Info),
  L (Verbindungsreihenfolge), O (zusätzliche OS-Verarbeitung erforderlich), G (Gruppe), T (TLS),
  C (komprimiert), x (unbekannt), o (betriebssystemspezifisch), E (ausgeschlossen),
  y (reiner Code), p (prozessorspezifisch)
```

Für jeden Abschnitt wird eine Zeile ausgegeben. Die Abschnitte ".strtab", ".shstrtab", ".symtab" und "NULL" sind ein integraler Bestandteil von ELF und immer vorhanden. Die Abschnitte ".debug" sind vorhanden, wenn die Quelle mit dem Flag "-g" zusammengestellt wurde. Der Abschnitt „.ARM.attributes“ definiert, für welchen ARM-Prozessor der enthaltene Code übersetzt wurde. Diese Abschnitte landen nicht auf dem Mikrocontroller. Die restlichen Abschnitte wurden im Linker-Skript definiert: ".VectorTable" enthält die Adressen der Ausnahmebehandlungsroutinen, ".text" enthält den Programmcode und die konstanten Daten für den Flash-Speicher, ".stack" den Stapel im RAM, ".data" enthält Variablen im RAM und ".bss" enthält null initialisierte Variablen im RAM. In diesen Abschnitten enthält die Spalte „Typ“ entweder „PROGBITS“ oder „NOBITS“, um anzuzeigen, ob der Abschnitt in der ELF-Datei tatsächlich Daten enthält. Dies gilt nur für „.VectorTable“, „.text“ und „. .Daten". Die Abschnitte ".bss" und ".stack" reservieren nur Speicher, der zur Laufzeit geschrieben wird. Die ELF-Datei enthält jedoch keine Daten, die in diese Abschnitte geschrieben werden müssen. Die Spalte „Adr“ definiert, wo dieser Abschnitt im Adressraum beginnt. Die nützlichste Spalte ist "Größe": Wenn Sie die Größen der Abschnitte ".VectorTable", ".text" und ".data" zusammenfassen, können Sie den verwendeten Flash-Speicher erhalten. Durch Summieren von ".data", ".stack" und ".bss" erhalten Sie die verbrauchte RAM-Größe. Beachten Sie, dass ".data" zweimal gezählt wird, da die Initialisierungsdaten in Flash gespeichert werden.

nm

Das Dienstprogramm "nm" druckt die in einer ELF-Datei definierten Symbole aus. Beispiel:

```
$ arm-none-eabi-nm prog1.elf
080001ec W ADC1_2_IRQHandler
20000404 b BlinkStep
08000192 t Blinktabelle
080001a3 t BlinkTableEnd
20000408 B _BssEnd
20000404 B _BssStart
```

Dies kann bei der Analyse von Fehlern in Linkerskripten hilfreich sein, bei denen Symbolen möglicherweise

falsche Adressen zugewiesen werden.

addr2line

Das Hilfsprogramm "addr2line" liest die Debug-Informationen aus einer ELF-Datei, um festzustellen, in welcher Zeile in welcher Quelldatei die Anweisung an einer bestimmten Adresse gefunden wurde. Zum Beispiel:

```
$ arm-none-eabi-addr2line 0x080000f0 -e prog1.elf
/tmp/test/prog1.S:24
```

Hier enthält Zeile 24 von "prog1.S" den Assembler-Befehl, der den Befehl erzeugt hat, der an der Adresse 0x080000f0 endet.

objcopy

Mit dem Dienstprogramm "objcopy" können Sie Programmdateien zwischen verschiedenen Formaten übersetzen. Es ist nützlich, die ELF-Dateien sowohl in das Intel Hex-Format als auch in eine einfache Binärdarstellung zu konvertieren. Zum Beispiel,

```
arm-none-eabi-objcopy -O ihex prog1.elf prog1.hex
```

Erzeugt eine „.hex“ -Datei, die ein Bild des Flash-Inhalts in hexadezimaler Form enthält. Mit

```
arm-none-eabi-objcopy -O binär prog1.elf prog1.bin
```

Es wird eine Binärdatei erstellt, die ein genaues 1: 1-Abbild des Flash-Inhalts enthält. Einige Flash-Tools erfordern diese Formate anstelle von ELF, und das Anzeigen der Binärdatei mit einem Hex-Editor kann ebenfalls interessant sein.

Schnittstelle zwischen C- und C ++ - Code

Da Assembler nur selten für die Implementierung komplizierter Projekte verwendet wird, aber meist nur für wenige zeitkritische oder besonders einfache Routinen, die Teil einer größeren Codebasis sind, die in einer Hochsprache geschrieben ist, ist die Verknüpfung von C- und Assembler-Code ein wichtiges Thema. was hier behandelt wird. Während es möglich ist, die Hauptprojektstruktur in Assembly zu schreiben und einige C-Module zu integrieren, geschieht dies normalerweise in umgekehrter Reihenfolge. Der größte Teil des gezeigten Codes ist bereits bereit, in C-Programme aufgenommen zu werden. Abgesehen von C ++ - Ausnahmen (nicht zu verwechseln mit ARM-Prozessor-Ausnahmen) funktioniert der Großteil dieses Themas in C ++ auf die gleiche Weise - diese werden jedoch auf eingebetteten Zielen sowieso selten verwendet.

Wenn Sie C, C ++ und Assembler-Code in einzelne .o-Objektdateien kompilieren, können Sie diese wie zuvor mit "ld" miteinander verknüpfen. Für C- und C ++ - Code ist jedoch in der Regel ein Zugriff auf die jeweilige Standardbibliothek erforderlich, und "ld" verknüpft diese standardmäßig nicht. Daher muss für einen Aufruf von "gcc" oder "g ++" für C oder "ld" eingesetzt werden C ++. Dies ruft intern "ld" auf und übergibt die erforderlichen Bibliotheken.

Einrichtung der Umgebung für C und C ++

Viele C-Projekte verwenden einen Reset-Handler und eine Vektortabelle, die in Assembly implementiert sind, obwohl das Schreiben in C ebenfalls möglich ist. Gemäß dem C-Standard beginnen C-Programme mit der Funktion „main ()“. Daher sollte der (Assembly-) Reset-Handler die Umgebung so einrichten, dass sie für C bereit ist, und dann „main“ aufrufen. Der C-Code kann dann später einige Assemblyfunktionen oder Inline-Assemblys aufrufen. Wenn Sie C ++ - Code oder eine GCC-Erweiterung für C-Code verwenden, müssen Sie einige zusätzliche Funktionen aufrufen, bevor Sie "main" aufrufen. Dies wird von C ++ verwendet, um die Konstruktoren globaler Objekte aufzurufen. Die C- und C ++ - Compiler geben eine Tabelle mit Funktionszeigern auf Funktionen aus, die beim Start aufgerufen werden sollen. Diese Tabelle muss in den Flash-Speicher verschoben werden, indem das Linker-Skript wie folgt geändert wird:

```
.text: {
*(.Text)
. = AUSRICHTEN (4);

_InitArrayStart =.;
* (SORT (.preinit_array *))
* (SORT (.init_array *))
```

```
_InitArrayEnd =.;
}> FLASH
```

Die Tabelle der Funktionszeiger ist sortiert, um die vom Compiler benötigte Reihenfolge beizubehalten. Die Symbole "_InitArrayStart" und "_InitArrayEnd" markieren den Anfang und das Ende dieser Tabelle. Ein Reset-Handler, der die Speicherinitialisierung wie zuvor ausführt und die Tabelle der Initialisierungsfunktionen aufruft, könnte folgendermaßen aussehen:

```
.syntax vereinheitlicht
.cpu cortex-m3
.Daumen

.Text
.type Reset_Handler,% Funktion
.global Reset_Handler
Reset_Handler:
ldr r0, = _DataStart
ldr r1, = _DataEnd
ldr r2, = _DataLoad

b 2f
1: ldr r3, [r2], # 4
str r3, [r0], # 4
2: cmp r0, r1
blo 1b

ldr r0, = _BssStart
ldr r1, = _BssEnd
ldr r2 = 0

b 2f
1: str r2, [r0], # 4
2: cmp r0, r1
blo 1b

ldr r4, = _InitArrayStart
ldr r5, = _InitArrayEnd

b 2f
1: ldr r0, [r4], # 4
blx r0
2: cmp r4, r5
blo 1b

bl main
1: bkpt
b 1b
.ltorg
```

Beachten Sie, dass zum Durchlaufen der Tabelle die Register r4 und r5 verwendet werden, da die aufgerufenen Funktionen diese möglicherweise nicht überschreiben. Die Anweisung "blx" wird benötigt, um den indirekten Funktionsaufruf auszuführen. Wenn alles eingerichtet ist, wird die Hauptfunktion aufgerufen. Bei eingebetteten Programmen sollte die Hauptfunktion niemals zurückkehren (d. H. Eine Endlosschleife enthalten). Wenn dies der Fall ist, ist dies ein Fehler. Um das Auffinden zu erleichtern, wird eine Endlosschleife mit einem erzwungenen Haltepunkt direkt nach dem Aufruf von "main" eingefügt.

Funktionen aufrufen

Um Assembly-Funktionen von C-Code aus aufzurufen und umgekehrt, sollten die Assembly-Funktionen die zuvor erwähnte Aufrufkonvention einhalten. C-Funktionen können wie Assembly-Funktionen aus Assembly-Code aufgerufen werden, indem die Parameter in Register r0-r3 und auf dem Stack abgelegt, die Funktion mit „bl“ aufgerufen und der Rückgabewert von r0 abgerufen wird. Um eine Assembly-Funktion aus C-Code aufzurufen, müssen Sie sie zunächst wie eine C-Funktion in C deklarieren. So rufen Sie beispielsweise eine Funktion auf, die 2 Ganzzahlargumente akzeptiert und eine Ganzzahl zurückgibt:

```
int AssemblyFunction (int a, int b);
```

Wenn Sie jetzt in Ihrem Assembly-Code eine Funktion mit dem Namen „AssemblyFunction“ definieren und über „.global“ exportieren, können Sie sie wie jede andere Funktion aus C-Code aufrufen.

Zugriff auf globale Variablen

Auf globale Variablen, die in C definiert sind, kann aus dem Assembly-Code unter Verwendung des Variablennamens wie auf Variablen zugegriffen werden, die im Assembly-Code definiert sind. Um über C-Code auf eine Assemblyvariable zuzugreifen, müssen Sie diese zunächst durch Angabe des Typs deklarieren. So deklarieren Sie beispielsweise eine Ganzzahlvariable:

```
extern int AssemblyVariable;
```

Wenn Sie jetzt in Ihrem Assembly-Code eine Variable mit dem Namen „AssemblyVariable“ definieren und über „.global“ exportieren, können Sie wie bei jeder Variablen auch über C-Code darauf zugreifen. Das "extern" ist erforderlich, um sicherzustellen, dass der C-Code nicht versucht, eine andere Variable mit demselben Namen zu deklarieren.

Uhrenkonfiguration

Standardmäßig verwenden STM32-Controller einen internen RC-Oszillator mit 8 MHz als Taktquelle für Core und Peripherie. Dieser Oszillator ist zu ungenau, um eine Uhr zu implementieren oder serielle Schnittstellen wie UART, USB oder CAN zu verwenden. Um eine genauere Uhr zu erhalten, wird üblicherweise ein äußerer Quarz angewendet. Viele STM32-Karten verfügen über einen 8-MHz-Quarz. Zur Verwendung ist ein Initialisierungscode erforderlich, der die integrierte Quarzoszillatorschaltung des Mikrocontrollers aktiviert und den Takteingang auf diesen umschaltet. Die STM32-Steuerungen enthalten auch eine PLL, die einen Eingangstakt mit einem konfigurierbaren Faktor multiplizieren kann, bevor er dem Prozessorkern und den Peripheriegeräten zugeführt wird. Auf diese Weise kann ein präziser und schneller Takt erzielt werden - der STM32F103 unterstützt eine Kernfrequenz von bis zu 72 MHz. Leider ist der Flash-Speicher nicht in der Lage, mit einer derart hohen Frequenz Schritt zu halten. Wenn Sie also eine schnelle Uhr aktivieren, muss der Flash-Speicher so konfiguriert werden, dass er abhängig von der Frequenz Wartezustände verwendet.

Die folgende Funktion konfiguriert die Flash-Wartezustände, aktiviert den Quarzoszillator, konfiguriert die PLL, um den Eingangstakt mit dem Faktor 9 zu multiplizieren, und verwendet diesen als Systemtakt. Der Vorteiler für den internen Bus APB1 ist auf 2 eingestellt. Unter der Annahme eines 8-MHz-Quarzes wird mit diesem Mikrocontroller die maximal mögliche Leistung erzielt - 72 MHz für den Kern und APB2-Bereich, 36 MHz für APB1. Wenn ein anderer Kristall verwendet wird, müssen die PLL-Faktoren angepasst werden.

```
RCC = 0x40021000

RCC_CR = 0x0
RCC_CR_PLLRDY = 25
RCC_CR_PLLON = 24
RCC_CR_HSERDY = 17
RCC_CR_HSEON = 16
RCC_CR_HSION = 0

RCC_CFGR = 0x04
RCC_CFGR_PLLMUL = 18
RCC_CFGR_USBPRE = 22
RCC_CFGR_PLLXTPRE = 17
RCC_CFGR_PLLSRC = 16
RCC_CFGR_PPRE2 = 11
RCC_CFGR_PPRE1 = 8
RCC_CFGR_HPRE = 4
RCC_CFGR_SWS = 2
RCC_CFGR_SW = 0

FLASH = 0x40022000
FLASH_ACR = 0
FLASH_ACR_PRFTBE = 4
FLASH_ACR_HLFCYA = 3
FLASH_ACR_LATENCY = 0

.type ConfigureSysClock,% -Funktion
.global ConfigureSysClock
ConfigureSysClock:
@ HSE einschalten
ldr r0, = RCC
ldr r1, = ((1 << RCC_CR_HSION) | (1 << RCC_CR_HSEON))
str r1, [r0, #RCC_CR]

@ PLL konfigurieren (aber noch nicht starten)
@ Mul = 9, Prediv = 1, APB1 Prescaler = 2, APB2 Prescaler = 1, AHB Prescaler = 1
```

```
ldr r2, = (((9-2) << RCC_CFGR_PLLMUL) | (1 << RCC_CFGR_USBPRE) | (1 << RCC_CFGR_PLLSRC) | (4 << RCC_CFGR_PPRE1))
str r2, [r0, #RCC_CFGR]

@ Wert für RCC_CR vorberechnen
orr r1, # (1 << RCC_CR_PLLON)


@ Warten Sie, bis HSE bereit ist
1: ldr r3, [r0, #RCC_CR]
und r3, # (1 << RCC_CR_HSERDY)
beq 1b

@ PLL einschalten
str r1, [r0, #RCC_CR]

@ Wert für RCC_CFGR vorberechnen
orr r2, # (2 << RCC_CFGR_SW)

@ Warten Sie, bis die PLL bereit ist
1: ldr r3, [r0, #RCC_CR]
und r3, # (1 << RCC_CR_PLLRDY)
beq 1b

@ Setzen Sie den Flash-Wartezustand auf 2
ldr r0, = FLASH
ldr r3, = ((1 << FLASH_ACR_PRFTBE) | (2 << FLASH_ACR_LATENCY))
str r3, [r0, #FLASH_ACR]
ldr r0, = RCC

@ Schalten Sie die Systemuhr auf PLL
str r2, [r0, #RCC_CFGR]

@ Wert für RCC_CR vorberechnen
bic r1, # (1 << RCC_CR_HSION)


@ Warten Sie, bis auf PLL umgeschaltet wurde
1: ldr r3, [r0, #RCC_CFGR]
und r3, # (3 << RCC_CFGR_SWS)
cmp r3, # (2 << RCC_CFGR_SWS)
bne 1b

@ Schalten Sie HSI aus, um Strom zu sparen
str r1, [r0, #RCC_CR]

bx lr
.ltorg
```

Viele Projekte führen die Uhrenkonfiguration durch den Reset-Handler durch, bevor sie die Hauptfunktion aufrufen. Wenn Sie dieser Praxis folgen möchten, platzieren Sie als erste Anweisung im „Reset_Handler“ einen „bl ConfigureSysClock“. Auf diese Weise wird das gesamte Setup mit der höheren Taktfrequenz ausgeführt, wodurch der Startvorgang beschleunigt wird. Dieser und der vollständige Startcode aus den vorherigen Kapiteln sind in der Datei "startup.S" im Beispielrepository implementiert. Wenn Sie es verwenden, geben Sie Ihren Code in die Hauptfunktion ein, in der RAM und Systemuhr bereits initialisiert sind. Dies wird im Beispiel „BlinkStartup“ gezeigt.

Projektvorlage & Makefile

Um Ihr eigenes Projekt schnell zu starten, finden Sie eine Projektvorlage im Beispielrepository unter dem Verzeichnis ProjectTemplate-STM32F103RB. Fügen Sie Ihren eigenen Anwendungscode in die program.S-Datei ein. Startup.S und Vectortable.S enthalten den Reset-Handler mit RAM-Initialisierung und die Vektortabelle mit Standard-Handler. Ein Linker-Skript ist ebenfalls enthalten.

Das Projekt enthält auch ein Makefile. Auf diese Weise können Sie Ihr Projekt schnell übersetzen, ohne die Assembler- und Linkerbefehle eingeben zu müssen. Einfach eintippen

```
machen
```

Um den Code zu übersetzen und program.elf-, program.bin- und program.hex-Dateien zu erstellen. Alle ".S" -Dateien im Verzeichnis werden automatisch übersetzt. Das Schreiben von Makefiles ist für sich genommen ein komplexes Thema mit vielen Informationen, die bereits im Web verfügbar sind. Daher werden hier keine

weiteren Erklärungen abgegeben.

Wikitext | Vorschau | Änderungen

Veröffentlichen | Abbrechen

Erweitert | Sonderzeichen | Hilfe

```
== Noch mehr Text ==
Hallo!
tag<br>

test6

[[Kategorie: ARM]] [[Kategorie: STM32]] [[Kategorie: Entwicklungstools]] [[Kategorie: Programmiersprachen]]
Die Prozessorarchitektur [[ARM]] ist in allen Industrieanwendungen weit verbreitet und auch eine bedeutende
Anzahl von Hobby- und Herstellerprojekten. In diesem Lernprogramm werden die Grundlagen der Programmierung
von ARM-Prozessoren in Assemblersprache vermittelt.

Tutorial von [[Benutzer: Erlkoenig | Niklas Gürtler]]. [https://www.mikrocontroller.net/topic/482409 Thread
im Forum] für Feedback und Fragen.

== Einleitung ==
=== Warum Assembler? ===
Heutzutage gibt es eigentlich keinen Grund, Assemblersprache für ganze Projekte zu verwenden, da
hochwertige Optimierungscompiler für Hochsprachen (insbesondere C und C ++) als kostenlose Open-Source-
Software leicht verfügbar sind und die ARM-Architektur speziell für Hochsprachen optimiert ist Sprachen.
Assembler-Kenntnisse sind jedoch nach wie vor hilfreich, um bestimmte Probleme zu debuggen, einfache
Software wie Bootloader und Betriebssystem-Kernel zu schreiben sowie Reverse Engineering-Software, für die
kein Quellcode verfügbar ist. Gelegentlich ist es erforderlich, einige leistungskritische Codeabschnitte
manuell zu optimieren. Manchmal wird behauptet, dass ARM-Prozessoren in Assembler nicht programmiert werden
können. Daher wird in diesem Tutorial gezeigt, dass dies sehr gut möglich ist, indem gezeigt wird, wie
ganze (kleine) Anwendungen vollständig in der ARM-Assemblersprache geschrieben werden!
```

Zusammenfassung:

/* Noch mehr Text */

Vorschau der Zusammenfassungszeile: (→Noch mehr Text)

☐ Nur Kleinigkeiten wurden verändert ☑ Diese Seite beobachten

Bitte beachte, dass alle Beiträge zu Mikrocontroller.net von anderen Mitwirkenden bearbeitet, geändert oder gelöscht werden können. Reiche hier keine Texte ein, falls du nicht willst, dass diese ohne Einschränkung geändert werden können.

Du bestätigst hiermit auch, dass du diese Texte selbst geschrieben hast oder diese von einer gemeinfreien Quelle kopiert hast (weitere Einzelheiten unter Mikrocontroller.net:Urheberrechte). **ÜBERTRAGE OHNE GENEHMIGUNG KEINE URHEBERRECHTLICH GESCHÜTZTEN INHALTE!**

Änderungen speichern | **Vorschau zeigen** | **Änderungen zeigen** | **Abbrechen**

▸ Profilingdaten des Parsers:

Kategorien: Seiten mit Syntaxhervorhebungsfehlern | ARM | STM32 | Entwicklungstools | Programmiersprachen