

Arbeitsweise und Umfeld des AVR Bootloaders optiboot

Karl-Heinz Kübbeler
`kh_kuebbeler@web.de`

14. Juni 2016

Inhaltsverzeichnis

1	Prinzipielle Arbeitsweise eines Bootloaders	3
2	Die Hardware der AVR 8-Bit Mikrocontroller	5
2.1	CPU und Speicherzugriff	5
2.2	Eingabe und Ausgabefunktion	7
2.3	Das Starten des AVR Mikrocontrollers	8
2.4	Das Beschreiben des AVR Speichers	10
2.4.1	parallele Programmierung	11
2.4.2	serieller Download mit ISP	11
2.4.3	Selbstprogrammierung mit serieller Schnittstelle	12
2.4.4	Diagnose Werkzeuge	13
3	Der optiboot Bootloader für AVR Mikrocontroller	14
3.1	Änderungen und Weiterentwicklung von Version 6.2	14
3.2	Automatische Größenanpassung in der optiboot Makefile	15
3.3	Zielvorgaben für optiboot Makefile	15
3.4	Die Optionen für die optiboot Makefile	17
3.5	Benutzung von optiboot ohne Bootloader-Bereich	18

Vorwort

Etwas intensiver habe ich mich mit den AVR-Bootloadern beschäftigt, als der Wunsch von Nutzern aufkam, die Transistortester-Software auf einigen Platinen der Arduino Familie zum Laufen zu bringen. Natürlich läuft die Software nicht als Arduino Sketch. Die Arduino Entwicklungsumgebung wird lediglich zur Darstellung von Ausgaben über die serielle Schnittstelle benutzt. Die Transistortester-Software benutzt auch nicht die Arduino Bibliothek. Das ist auch gar nicht notwendig, um den Bootloader zu benutzen.

Der Bootloader ist ein kleines Programm, welches Programm-Daten über eine serielle Kommunikation von einem Host (PC) entgegennehmen kann und in den Arbeitsspeicher des Mikrocontrollers laden kann. Da die Transistortester-Software ziemlich viel Programmspeicher braucht, sollte der Bootloader nur wenig vom Programmspeicher für sich selbst belegen. Außer dem Programmspeicher sollte der Bootloader auch den anderen nicht flüchtigen Speicher des AVR beschreiben können, das EEprom. Damit war die Zielsetzung klar. Es sollte ein Bootloader her, der auch das Beschreiben des EEproms unterstützt, aber wenig Flash-Speicherplatz benötigt.

Kapitel 1

Prinzipielle Arbeitsweise eines Bootloaders

Ein Bootloader ist ein kleines Programm, welches neue Programm-Daten für einen Prozessor über eine Datenschnittstelle in Empfang nehmen kann und in den Arbeitsspeicher dieses Prozessors ablegen kann. Üblicherweise wird dieses über die Datenschnittstelle empfangene Programm nach Beenden der Übertragung vom Bootloader gestartet. Damit ist ein Rechner mit einem beschreibbaren Arbeitsspeicher dann in der Lage, beliebige Anwenderprogramme aus dem Arbeitsspeicher auszuführen.

Im Prinzip ist damit das BIOS eines PC's auch ein Bootloader, allerdings erweitert um Möglichkeiten, die Schnittstelle für den ersten Zugriff auf Programm-Daten einzustellen. Beim PC läßt sich so oft eine ganze Kette von Peripheriegeräten einstellen, die auf das Vorhandensein von Programm-Daten getestet werden. Beim PC wird dann oft eine zweite Stufe gestartet, die weitere Einstellungen (Auswahl von Betriebssystemen oder abgesicherter Modus) ermöglicht.

Bei Mikrocontrollern ist der Bootloader meistens einfacher gestaltet. Hier wird nur eine Schnittstelle untersucht und es gibt auch keine weitere Einstellmöglichkeit im Betrieb. Ein Merkmal für die Betriebsweise des Bootloaders besteht übrigens im Typ des Arbeitsspeichers. Wenn der Arbeitsspeicher des Rechners aus flüchtigem Speicher besteht (RAM = Random Access Memory), muß der Bootloader vor einem Anwender-Programmstart sicher sein, daß er gerade vorher ein Programm selbst geladen hatte.

Bei einem Mikrocontroller mit nicht flüchtigem Arbeitsspeicher (Flash) darf der Bootloader annehmen, daß bereits irgendwann einmal vorher ein Anwender-Programm in den Speicher geladen wurde. Deshalb wird nach einer angemessenen Wartezeit auf neue Programm-Daten versucht, das Anwenderprogramm zu starten. Dabei ist es egal, ob gerade vorhin ein neues Anwenderprogramm geladen wurde oder nicht. Selbst wenn noch nie ein Anwenderprogramm geladen wurde oder ein fehlerhaftes, passiert nichts Schlimmes. Die Möglichkeit, ein neues Anwenderprogramm zu laden, bleibt ja weiterhin erhalten. Es ist eher das Gegenteil der Fall. Durch das fehlenden Anwenderprogramm versucht der Bootloader immer wieder, die Kommunikation über die serielle Schnittstelle aufzubauen.

Die Abbildung 1.1 zeigt die prinzipielle Arbeitsweise von Bootloadern, die ihre Daten über eine serielle Schnittstelle empfangen.

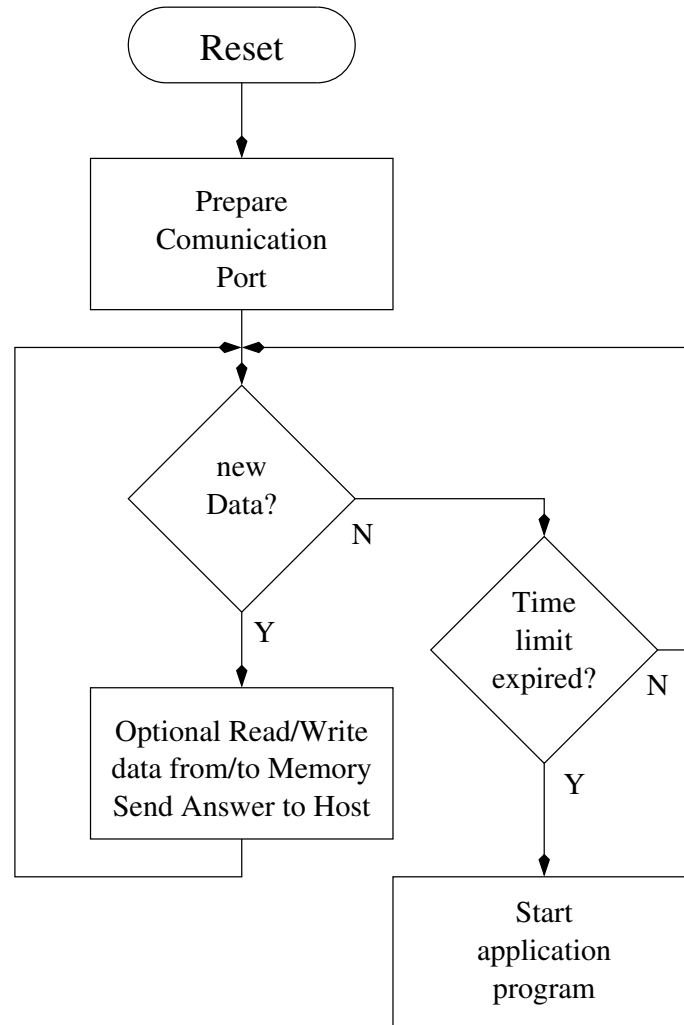


Abbildung 1.1. Prinzipielle Arbeitsweise eines Bootloaders

Der AVR-Zielprozessor wird beim Start des Daten-Sendeprozess auf dem PC zurückgesetzt. Wenn dies nicht automatisch erfolgt, muß der AVR-Prozessor von Hand zurückgesetzt werden. Der PC versucht die Kommunikation mit dem AVR-Prozessor aufzubauen, indem er ein Datenbyte über die serielle Schnittstelle schickt und auf die Antwort des AVR-Prozessors wartet. Wenn die Antwort nicht in angemessener Zeit erfolgt, wird dieser Vorgang wiederholt. Das Bootloader Programm auf dem AVR Prozessor wartet nur eine begrenzte Zeit auf Daten (Time limit). Beim Überschreiten der Wartezeit wird versucht, das Anwender-Programm im Flash-Speicher zu starten.

Kapitel 2

Die Hardware der AVR 8-Bit Mikrocontroller

2.1 CPU und Speicherzugriff

Auf dem Chip der AVR 8-Bit Mikrocontroller ist alles vereinigt, was ein digitaler Minirechner zum Arbeiten braucht. Es ist ein Taktgenerator, Register, Arbeitsspeicher (RAM), Programmspeicher (Flash), Eingaberegister und Ausgaberegister vorhanden. Der Inhalt von Registern und Arbeitsspeicher geht bei jedem Neustart verloren. Der Inhalt des Programmspeicher (Flash) und auch des meistens zusätzlich vorhandenem Datenspeicher (EEPROM) bleiben aber erhalten. Die Abbildung 2.1 zeigt ein vereinfachtes Blockdiagramm eines 8-Bit AVR Mikrocontrollers.

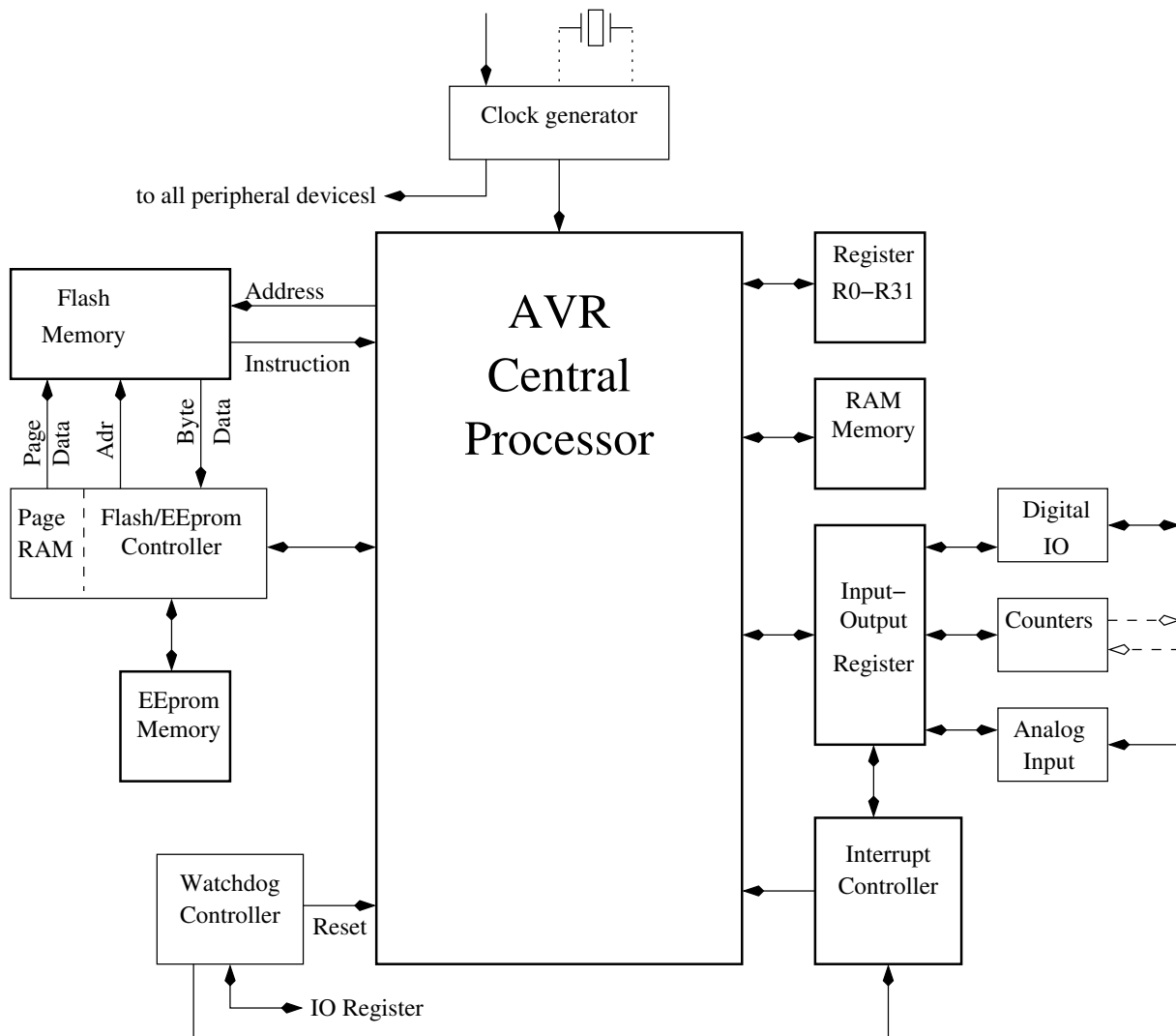


Abbildung 2.1. Vereinfachtes Blockschaltbild eines AVR-Mikrocontrollers

Aus dem Diagramm ist zu entnehmen, daß die CPU (Central Processor Unit) sehr leicht auf die Register R0-R31 und auf den RAM-Speicher zugreifen kann. Auch der Zugriff auf die Eingabe (Input) oder Ausgabe (Output) Register ist leicht möglich. Der Zugriff auf den Programmspeicher (Flash) ist dagegen nur über den zugehörigen Controller möglich und deutlich komplizierter.

Nur die eigentliche Befehls-Ausführungseinheit kann einfach auf die Daten des Flash Speichers für die eingestellte Programm-Adresse zugreifen. Mit dem Load Immediate Befehl (LDI) können dann auch Bestandteile des 16-Bit Befehls-Wortes zu den oberen Registern (R16-R31) transferiert werden. Auch bei den Befehlen ADIW, ANDI, CPI, ORI, SBCI, SBIW und SUBI werden Teile des 16-Bit Befehls-Wortes verarbeitet.

Normalerweise wird bei der Programmausführung die Flash-Adresse je nach Befehls-Größe um ein Wort oder zwei Worte erhöht. Eine Ausnahme hiervon sind bei der normalen Programm-Abarbeitung nur die bedingten oder unbedingten (RJMP, JMP, IJMP, RCALL, CALL, ICALL, RET, RETI) Sprung-Anweisungen.

Außerdem kann ein Reset Ereignis oder ein Unterbrechungs-Signal (Interrupt) für eine Abweichung von der Regel führen. Beim Reset wird der Prozessor zurückgesetzt und der Programm-Zähler auf eine vorher eingestellte Adresse gesetzt. Bei einem Interrupt wird der Programm-Zähler auf die dem Ereignis zugehörige Adresse einer Sprungtabelle gesetzt. Normalerweise ist die Start-Adresse für das Reset Ereignis die Adresse 0. Für Bootloader-Zwecke kann dies aber bei vielen AVR-Prozessoren mit speziellen Konfigurations-Bits (Fuse) anders eingestellt werden.

Ein wahlfreier Lesezugriff zum Programmspeicher ist nur über den Flash-Controller möglich.

Dabei muß dem Controller erst die gewünschte Byte-Adresse mitgeteilt werden. Erst dann ist ein Lesen des Datenbytes mit einem Spezial-Befehl möglich.

Noch komplizierter wird es beim Schreibzugriff. Der Schreibzugriff des Programmspeichers ist nur seitenweise möglich. Die Flash-Speicherseite muß vor jedem Beschreiben vorher gelöscht sein. Die neuen Daten für eine Flash-Seite müssen vor dem Beschreiben komplett in den Zwischenspeicher des Flash-Controllers geladen werden. Erst dann kann der Controller mit dem Neubeschreiben der Flash-Seite beauftragt werden. Dieses Verfahren läßt sich anschaulich mit dem Bedrucken einer Seite mit einem Stempel vergleichen. Der Stempel kann mit austauschbaren Zeichen bestückt werden, so daß jeder Text möglich ist. Für das Stempeln der Information muß aber eine leere Papierseite vorhanden sein und der Stempel muß zuerst mit den Zeichen für den Text bestückt sein. Erst dann kann mit einem Stempeldruck die leere Seite neu beschrieben werden.

Auch der EEPROM Speicher (nicht flüchtiger Speicher für Daten) wird über einen speziellen Controller zugegriffen. Hier ist das Beschreiben des Speichers aber etwas einfacher, ist aber auch nur über einen Controller möglich. Der EEPROM-Controller kann nicht zusammen mit dem Flash-Controller benutzt werden, da die Controller wohl gemeinsame Teile haben.

2.2 Eingabe und Ausgabefunktion

Über die IO-Register kann die CPU auf die externen Pins zugreifen. Die Register sind Byte weise (8-Bit) organisiert, so daß mit einem Register bis zu 8 Pins angesprochen werden können. Die Abbildung 2.2 zeigt den Aufbau einer Port Pin Beschaltung.

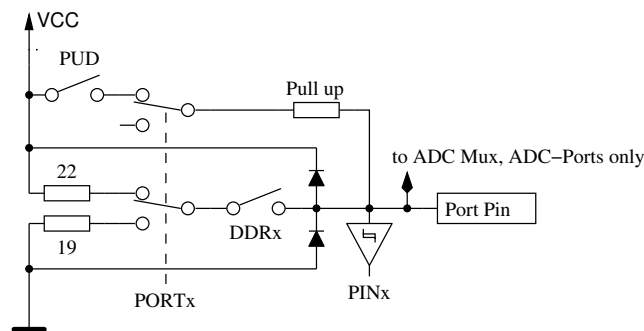


Abbildung 2.2. Vereinfachtes Schaltbild jedes AVR-Portpins

Jeder Pin ist mit einem Port Bit fest verknüpft und kann normalerweise sowohl als Ausgangs-Pin (PORT) als auch als Eingangs-Pin (PIN) verschaltet werden. Zur Umschaltung der Funktion der 8 Bits eines Ports dienen die 8 Bits des Data Direction Registers DDR. Es gibt also für jeden Pin eine zugeordnete Bitnummer in drei verschiedenen Registern. Mit dem DDR Register kann die Datenrichtung für jedes des zugeordneten Pins umgeschaltet werden. Das PIN Register gibt den Zustand der Spannung an den 8 Eingangspins an. Unterhalb etwa der halben Betriebsspannung wird eine 0 angezeigt, darüber eine 1. Wenn das dem Pin zugehörige DDR Bit auf Ausgang (1) geschaltet ist, wird mit dem zugehörigen PORT Bit der Ausgang auf 0 V oder auf Betriebsspannung (VCC) geschaltet. Bei mit dem auf 0 gesetzten DDR Bit inaktiviertem Ausgang wird je nach Zustand des zugehörigen PORT Bits ein Pull-Up Widerstand für diesen Pin zugeschaltet (PORT Bit = 1) oder nicht (PORT Bit = 0). Wenn allerdings das PUD Bit im MCU Konfigurations-Register MCUCR gesetzt ist, bleiben alle internen Pull-Up Widerstände inaktiv. Alle zugehörigen Register und Bits einer Gruppe (8-Bit) benutzen immer den gleichen Kennbuchstaben. Also für die zweite Gruppe wäre der Buchstabe ein B. Der Ausgabe Port würde also PORTB heißen, der Eingangs Port würde PINB heißen und das Register für die Datenrichtung DDRB. Für die Bezeichnung der einzelnen Bits wird jeweils eine Ziffer 0-7 angehängt. Zum Beispiel würde das Bit 0 des Eingangs Ports dann PINB0

heißen. Diese Bezeichnungsweise wird so in der Beschreibung der Hardware von Atmel verwendet und wird üblicherweise auch innerhalb von Programmiersprachen so verwendet.

2.3 Das Starten des AVR Mikrocontrollers

In der vom Werk ausgelieferten Konfiguration des Mikrocontrollers löst das erste Erreichen der erforderlichen Betriebsspannung einen Reset des Prozessors aus. Dabei werden die IO-Register auf vorbestimmte Werte gesetzt, noch etwas Zeit zur Stabilisierung der Betriebsspannung gewartet und dann der Befehl ausgeführt, der auf der Adresse 0 des Flash-Speichers steht. In aller Regel sind die Ausgabe-Funktionen aller Pins in diesem Zustand abgeschaltet. Außer diesem Einschalt-Reset gibt es noch drei weitere Gründe für einen Reset des Prozessors. Der Grund für den Reset wird in einem MCU Status-Register (MCUSR) mit 4 Bits festgehalten.

Name des Flags	Grund für den Reset
PORF	Power-on Reset Dieser Reset wird beim ersten Erreichen der Betriebsspannung ausgelöst. Dieser Grund kann nicht abgeschaltet werden.
BORF	Brown-out Reset Der Reset wegen Spannungseinbruch kann nur dann vorkommen, wenn die Funktion mit den BODLEVEL Bits in dem Fuse-Bit freigeschaltet ist und kein Brown-out Interrupt benutzt wird.
EXTRF	External Reset Wird ausgelöst durch Pegel 0 am Reset Pin, wenn RSTDISBL nicht konfiguriert wurde.
WDRF	Watchdog Reset Wird nur ausgelöst, wenn der zugehörige Interrupt nicht freigeschaltet wurde.

Tabelle 2.1. Die verschiedenen Reset Gründe im MCUSR Register

Durch Setzen der entsprechenden Konfigurations-Bits in den Fuses des AVR-Mikrocontrollers kann die Start-Adresse abweichend von 0 eingestellt werden. Die Abbildung 2.3 zeigt die Möglichkeiten für den ATmega168. Dieser Prozessor hat insgesamt 16384 Byte Arbeitsspeicher (Flash). Der Befehls-Interpreter des Mikrocontrollers greift übrigens auf einen 16-Bit breiten Befehlscode des Flash Speicher zu. Deshalb ist der höchste Programm-Zähler Wert nur 8190! Das es nicht 8192 ist, ist klar, weil ja die Zählung bei 0 beginnt, aber es ist noch ein Wort weniger, weil das letzte Wort im Flash Speicher für die Versionsnummer von Optiboot verwendet wird.

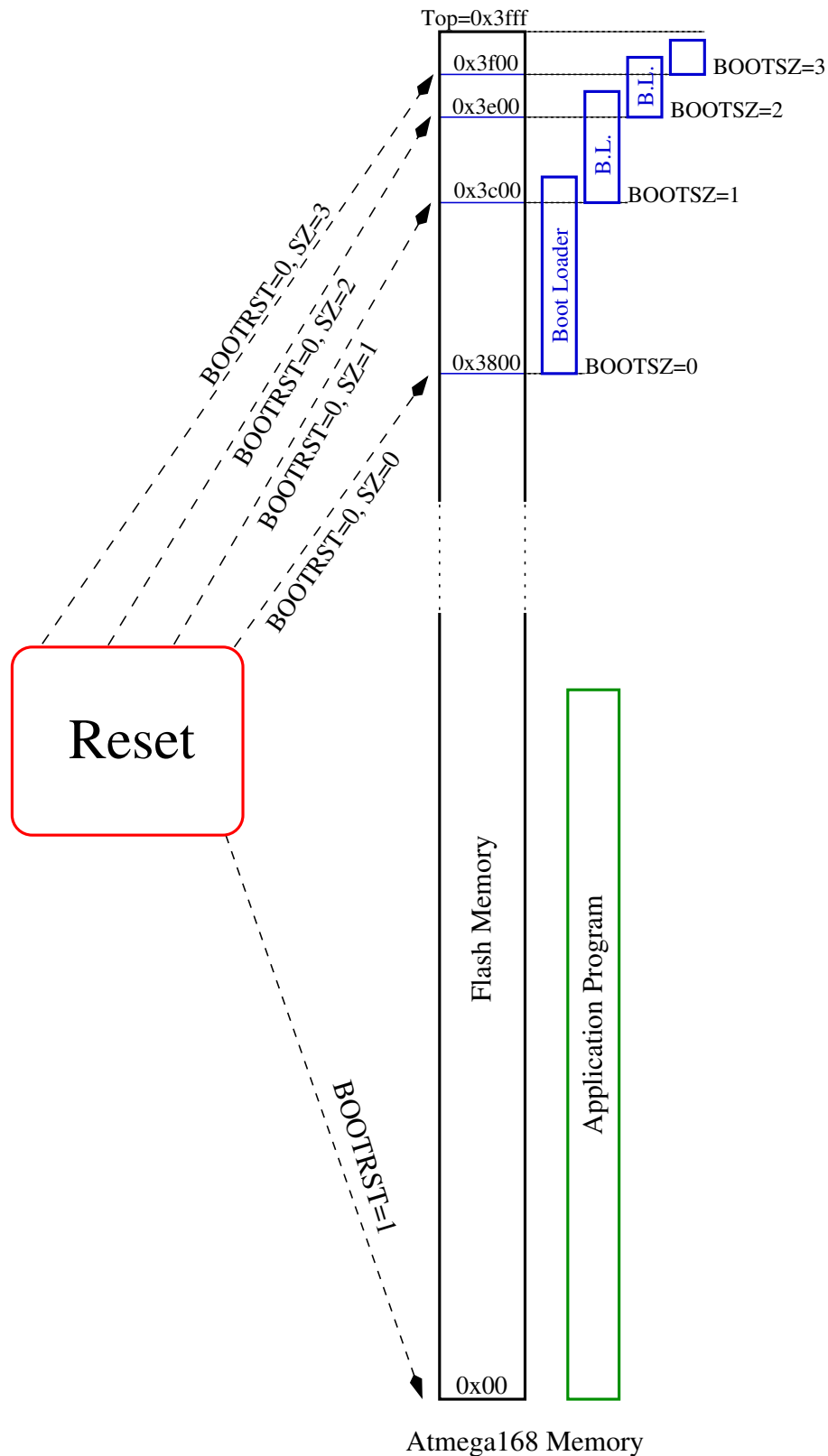


Abbildung 2.3. Verschiedene Start-Möglichkeiten für den ATmega168

Als mögliche Bootloader-Größen unterstützt der ATmega168 256 Bytes (BOOTSZ=3), 512 Bytes (BOOTSZ=2), 1024 Bytes (BOOTSZ=1) und 2048 Bytes (BOOTSZ=0). Für das Anwenderprogramm ist natürlich interessant, den Bootloader-Bereich möglichst klein zu wählen, damit möglichst viel Speicherplatz für das Anwenderprogramm zur Verfügung steht. Der Bootloader wird ja immer auf die höchstmögliche Start-Adresse platziert. Durch Programmieren der Lock-Bits des AVR-Mikrocontrollers läßt sich der Bootloader-Bereich gegen Überschreiben sichern. Die einmal installierte

Sicherung des Flash-Speichers lässt sich dann aber nur durch komplettes Löschen (erase) des AVR Speichers wieder zurücksetzen.

Für diesen Prozessor, wie auch für den Mega48 und Mega88, befinden sich die Steuerbits für den Bootloader Start in der extended Fuse. Dies gilt auch für die BOOTRST Fuse, mit der die Startadresse von 0 auf den Bootloader Start umgeschaltet wird. Für viele andere AVR Mikrocontroller, auch dem ATmega328, befinden sich die gleichen Steuerbits in der high Fuse. Die Tabelle 2.2 zeigt die Speichergrößen für verschiedene AVR-Mikrocontroller sowie die Möglichkeiten für den Bootloader-Bereich. Die verwendeten Bit-Nummern für die Bootloader Einstellung sind übrigens die gleichen, egal ob high oder extended Fuse Register.

Processor Typ	Flash Größe	EEProm Größe	RAM Größe	UART	Boot Config Fuse	BOOTSZ			
						=3	=2	=1	=0
ATmega48	4K	256	512	1	Ext.	256	512	1K	2K
ATtiny84	8K	512	512	-	-	(N x 64)			
ATmega8	8K	512	1K	1	High	256	512	1K	2K
ATmega88	8K	512	1K	1	Ext.	256	512	1K	2K
ATmega16	16K	512	1K	1	High	256	512	1K	2K
ATmega168	16K	512	1K	1	Ext.	256	512	1K	2K
ATmega164	16K	512	1K	1	High	256	512	1K	2K
ATmega32	32K	1K	2K	1	High	512	1K	2K	4K
ATmega328	32K	1K	2K	1	High	512	1K	2K	4K
ATmega324	32K	1K	2K	2	High	512	1K	2K	4K
ATmega644	64K	2K	4K	2	High	1K	2K	4K	8K
ATmega640	64K	4K	8K	4	High	1K	2K	4K	8K
ATmega1284	128K	4K	16K	2	High	1K	2K	4K	8K
ATmega1280	128K	4K	8K	4	High	1K	2K	4K	8K
ATmega2560	262K	4K	8K	4	High	1K	2K	4K	8K

Tabelle 2.2. Bootloader Konfigurationen für verschiedene Mikrocontroller

Übrigens funktioniert ein Bootloader selbst dann beim ersten Mal, wenn das BOOTRST Fuse Bit nicht gelöscht wurde. Dann steht der Reset-Vektor immer noch auf Adresse 0, auf den normalerweise das Anwenderprogramm steht. Da aber noch kein Anwenderprogramm geladen wurde, läuft die CPU durch den Flash Speicher, bis er auf den Bootloader-Code trifft. Bei diesem Prozessor sind das weniger als 8000 Befehle, die die CPU bei 8 MHz Taktrate in weniger als 1 ms durchläuft. Sobald aber ein Anwenderprogramm geladen wurde, startet mit einem Reset immer das Anwenderprogramm, solange das BOOTRST Bit gesetzt bleibt. Der Bootloader würde mit gesetztem BOOTRST Bit nicht mehr funktionieren, weil er gar nicht mehr angesprochen wird.

2.4 Das Beschreiben des AVR Speichers

Die AVR Mikrocontroller kennen 3 verschiedene nicht flüchtige Speicher. Der wichtigste ist der Programmspeicher, der sogenannte Flash-Speicher.

Daneben gibt es noch einige Konfigurationsbits, mit denen die Eigenschaften des Prozessors eingestellt werden können. Diese sind Konfigurationsbits sind aufgeteilt in mehrere Bytes, die lfuse (low Fuse), hfuse (high Fuse), efuse (extended Fuse), das lock-Byte und das Kalibrations-Byte. Das Kalibrations-Byte dient dem Frequenz-Abgleich des internen RC-Oszillators. Mit dem lock-Byte können Zugriffe auf die Speicher gesperrt werden. Beim lock-Byte lassen sich einmal aktivierte Bits nicht

wieder zurücksetzen. Der einzige Weg, einmal gesetzte lock-Bits wieder zurückzusetzen, ist ein komplettes Löschen aller AVR-Speicher. Beim lock-Byte werden die Schutzfunktionen durch Löschen von entsprechenden Bits auf 0 aktiviert. Im gelöschten Zustand des AVR-Mikrocontrollers sind alle wirk-samen Bits des lock-Bytes auf 1 gesetzt. Die Aufteilung des Konfigurationsbits auf die verschiedenen Fuse-Bytes unterscheidet sich für die verschiedenen Prozessor-Modelle und sollte im jeweiligen Da-tenblatt nachgelesen werden. Mit den Konfigurationsbits lassen sich Art der Taktgewinnung für den Prozessor, eine Startverzögerung und eine Betriebsspannungsüberwachung einstellen.

Die meisten AVR Mikrocontroller verfügen auch über einen nicht flüchtigen Datenspeicher, das EEprom. Dieser Speicher hat keine spezielle Aufgabe für den Prozessor. Es ist lediglich eine Mög-lichkeit, Daten für den nächsten Programmstart festzuhalten.

2.4.1 parallele Programmierung

Alle drei Speicherarten können mit verschiedenen Methoden beschrieben und auch gelesen werden. Normalerweise wird die parallele Methode zum Beschreiben der nicht flüchtigen Speicher eher selten verwendet. Manchmal hilft diese Methode aber Prozessoren wieder zu beleben, die mit anderen Me-thoden nicht mehr ansprechbar sind. Beispielsweise funktioniert die serielle Programmierung dann nicht mehr, wenn mit den Fuse-Bits die Funktion des Reset-Pins abgeschaltet wurde. Diese Methode ist daran zu erkennen, daß bei der Programmierung der Reset Eingangspin auf höhere Spannungs-werte (12V) gesetzt wird. Deshalb heißt diese Methode auch HV-Programmierung. Einzelheiten zu dieser Programmierung sollten dem jeweiligen Datenblatt entnommen werden.

2.4.2 serieller Download mit ISP

Der normale Weg zum Programmieren des nicht flüchtigen Speichers ist die serielle Programmierung. Die Atmel Dokumentation spricht auch von Serial Downloading. Dabei wird eine SPI (Serial Parallel Interface) Schnittstelle verwendet. Die SPI Schnittstelle besteht aus drei Signalen, MOSI, MISO und SCK. Damit der Prozessor überhaupt diese spezielle Art der Programmierung benutzt, ist außerdem das Reset Signal erforderlich, das auf 0 gehalten werden muß. Zusammen mit den Betriebsspannungs Signalen GND und VCC (etwa 2.7V bis 5V) ergibt sich eine Schnittstelle, die oft bereits auf einer Platine integriert ist, die ISP (In System Programming) Schnittstelle. Die Abbildung 2.4 zeigt zwei gebräuchliche Stecker, die auf Platinen mit AVR Mikrocontrollern integriert sind.

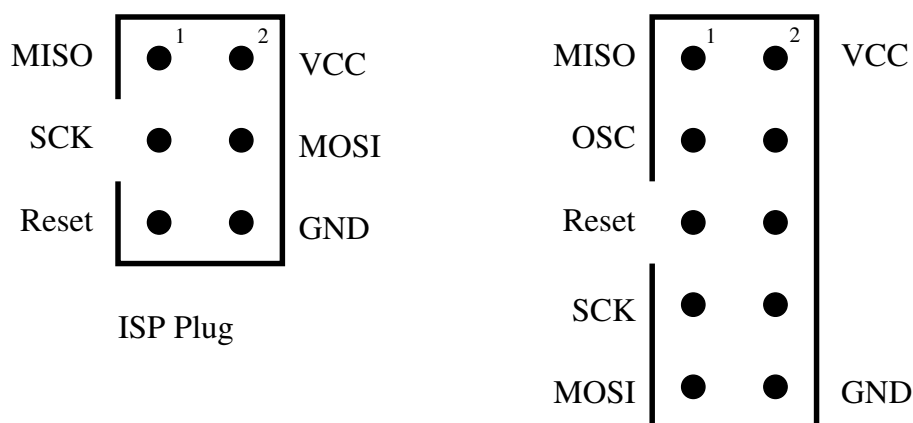


Abbildung 2.4. Zwei verschiedene Arten von ISP Steckern

Bei der 10-poligen Variante des ISP-Steckers kann zusätzlich das Takt Signal OSC für die Ver-sorgung des AVR mit einem Takt aufgelegt sein. Einer dieser beiden Stecker ist also in der Re-gel erforderlich, um einen neuen Bootloader in den Flash-Speicher eines AVR-Mikrocontrollers zu

schreiben. Die Daten für den Flash-Speicher werden im allgemeinen auf einem PC erzeugt. Für das Transferieren der Daten zum AVR-Mikrocontroller ist ein ISP-Programmer erforderlich, der meistens zur Rechnerseite hin über einen USB-Anschluss verfügt. Es ist aber auch ein Anschluss des ISP-Programmers über eine serielle oder parallele Schnittstelle möglich. Die USB-Schnittstelle hat aber den Vorteil, daß auch die Stromversorgung (5V oder 3.3V) für dem Mikrocontroller direkt über die Schnittstelle zur Verfügung gestellt wird. Es gibt auf dem Markt eine Auswahl an solchen ISP-Programmern, von Atmel wird beispielsweise der AT AVR-ISP MK2 Programmer angeboten. Auch die Verwendung eines Arduino UNO oder eines anderen Arduino ist mit Spezialprogramm für eine angeschlossene ISP-Schnittstelle möglich. Ich selbst verwende einem DIAMEX ALL-AVR, der über beide ISP-Steckervarianten verfügt und einige Zusatzfunktionen hat.

2.4.3 Selbstprogrammierung mit serieller Schnittstelle

Da der AVR-Prozessor den Flash und EEPROM Speicher auch selbst beschreiben kann, kann man mit einer der beiden anderen Methoden ein kleines Programm in den Flash-Speicher laden, der Daten über eine serielle Schnittstelle empfängt und diese Daten in den Flash- oder auch EEPROM-Speicher schreibt. Genau das macht der Bootloader Optiboot. Das Setzen der Fuses oder des Lock-Bytes ist mit dieser Methode oft nicht möglich und wird auch vom Bootloader nicht unterstützt. Die Fuses und das Lock-Byte müssen also immer mit einer der anderen Methoden geschrieben werden. Für die Übermittlung der Daten werden Teile des STK500 Communication Protocol von Atmel verwendet. Da moderne Rechner meist nicht mehr über serielle Schnittstellen verfügen, wird heute meistens ein USB - Seriell UART Wandler wie beispielsweise das FTDI Chip von Future Technology Devices International Ltd verwendet. Ein Modul mit diesem Wandler-Chip ist z.B. das UM232R. Die Chips PL2303 von Prolific Technology Inc. und CP2102 von Silicon Laboratories Inc. erfüllen wohl den gleichen Zweck. Von Atmel kann ein entsprechend programmierter ATmega16U2 auch diese Aufgabe übernehmen. Alle Chips haben eine einstellbare Baud-Rate und haben TTL-Pegel für die Signale. Für echte RS232 Signale müßten noch Pegelwandler verwendet werden. Die sind aber für die AVR Mikrocontroller nicht notwendig. Einer dieser Chips ist auf den Arduino Platinen mit USB-Schnittstelle integriert. Für eine schnelle Reaktion auf die Programm-Datenübertragung sollte das DTR-Signal des Schnittstellenwandlers über einen 100nF Kondensator mit dem Reset Eingang des AVR-Prozessors verbunden sein. Die Abbildung 2.5 zeigt eine typische Anschlussart.

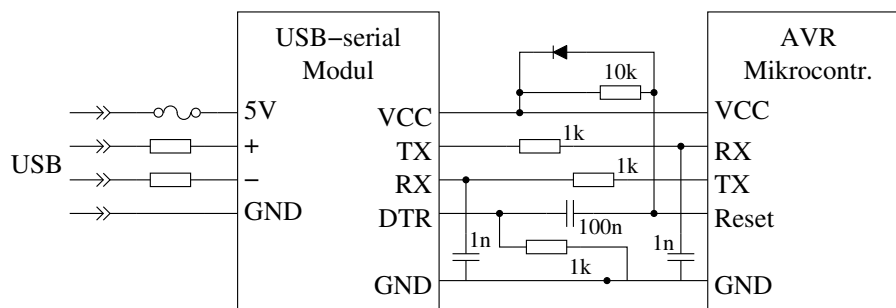


Abbildung 2.5. Anschluss eines USB-seriell Wandlers an den Mikrocontroller

Bei der Verwendung von USB-seriell Modulen sollte auf die richtige Wahl der Betriebsspannung geachtet werden. Viele Module können auf 3.3V oder 5V Signalpegel gestellt werden (Jumper). Wenn man einen Arduino UNO mit gesockeltem ATmega328p besitzt, kann man den ATmega328p entfernen und das Board als USB-seriell Wandler benutzen. So kann man dann auch die Programmdateien zu einem anderen AVR-Prozessor übertragen. Bei häufigem Gebrauch ist natürlich ein separates USB-seriell Modul sinnvoll. Die beiden 1 nF Kondensatoren an den RX-Eingängen filtern Impulsspitzen weg. Der Test des Software-UART Programms war nur erfolgreich mit einem kleinen Kondensator am

RX Eingang des AVR Prozessors. Die Tastspitze eines Oszilloskops war aber als „Filter“ schon ausreichend. Die Hardware-UART Schnittstelle filtert wohl besser und läuft mit und ohne Kondensator einwandfrei.

Bei der Arduino Entwicklungsumgebung Sketch ist unter „Tools - serieller Port“ die Möglichkeit vorgesehen, eine serielle Schnittstelle auszuwählen. Weiter kann dann unter „Tools - Serial Monitor“ ein Kontrollfenster geöffnet werden, welches die seriellen Ausgaben des AVR Mikrocontrollers auf dem Bildschirm darstellen kann. Die Baud-Rate der seriellen Schnittstelle kann in diesem Fenster eingestellt werden. Der laufende Serial Monitor des Arduino Sketch stört aber das Laden von Programmen über die serielle Schnittstelle, wenn das gleiche USB-Seriell Wandlermodul verwendet wird. Es gibt aber die Möglichkeit, ein weiteres USB-Seriell Modul in den PC einzustecken und bei diesem nur das RX-Signal an den TX-Port des AVR-Mikrocontrollers anzuschließen. Dieser zweite serielle Eingang horcht so die Ausgaben des AVR mit und stört dabei die serielle Kommunikation beim Programm-laden nicht, wenn diese Schnittstelle für den Serial Monitor gewählt wird.

2.4.4 Diagnose Werkzeuge

Unter dem Linux-Betriebssystem gibt es ein Werkzeug (Tool) mit dem seltsamen Namen jpnevulator, mit dem zwei serielle Eingänge zur selben Zeit überwacht werden können. Die übertragenen Daten werden im sedezimal (hex) Format dargestellt und mit der Option -a auch als ASCII-Zeichen. Mit der Option -timing-print werden die System-Zeiten der empfangenen seriellen Datenpakete gezeigt. Um eine Beeinflussung des Datenverkehrs beim Laden von Programmdateien auszuschließen, sollten zwei separate USB-Seriell Module für die Überwachung benutzt werden. Der serielle Eingang eines Moduls wird an das TX-Signal und der serielle Eingang des anderen Moduls an das RX-Signal des AVR-Mikrocontrollers angeschlossen. Zusammen mit dem Modul für das Programmladen sind dann drei solche Module an den PC angeschlossen. Natürlich müssen alle drei Module auf die gleiche Baudrate eingestellt sein (stty ... -F /dev/ttyUSB1). Der komplette Aufruf für das Werkzeug mit dem Beginn der Ausgaben könnte beispielsweise so aussehen:

```
jpnevulator -a --timing-print --read --tty "/dev/ttyUSB1" --tty "/dev/ttyUSB2"
2016-05-29 11:05:06.589614: /dev/ttyUSB0
30 20                                0
2016-05-29 11:05:06.589722: /dev/ttyUSB1
14 10                                ..
2016-05-29 11:05:06.593593: /dev/ttyUSB0
41 81 20                            A.
2016-05-29 11:05:06.594581: /dev/ttyUSB1
14 74 10                            .t.
2016-05-29 11:05:06.597583: /dev/ttyUSB0
41 82 20                            A.
2016-05-29 11:05:06.598574: /dev/ttyUSB1
14 02 10                            ...
2016-05-29 11:05:06.601586: /dev/ttyUSB0
42 86 00 00 01 01 01 03 FF FF FF FF 00 80 04 B.....
00 00 00 80 00 20                    .....
2016-05-29 11:05:06.603608: /dev/ttyUSB1
14 10                                ..
2016-05-29 11:05:06.605639: /dev/ttyUSB0
45 05 04 D7 C2 00 20                E.....
2016-05-29 11:05:06.606576: /dev/ttyUSB1
14 10                                ..
...
```

Kapitel 3

Der optiboot Bootloader für AVR Mikrocontroller

Der optiboot Bootloader wurde in der Sprache C von Peter Knight und Bill Westfield entwickelt. Die Version 6.2 habe ich als Basis für die hier beschriebene überarbeitete Assembler Version benutzt. Dabei möchte ich betonen, daß ich den optiboot Bootloader nicht neu erfunden habe, sondern lediglich weiter optimiert. Viele Anpassungen an verschiedene Zielprozessoren und spezielle Platinenentwürfe waren bereits in der Version 6.2 vorhanden. Es werden Teile des STK500 Kommunikations-Protokolls benutzt, die in der AVR061 [7] von Atmel veröffentlicht wurde.

3.1 Änderungen und Weiterentwicklung von Version 6.2

Im wesentlichen habe ich das komplette Programm in die Assembler- Sprache umgeschrieben und die Makefile so angepasst, daß die Programmlänge automatisch weiterverarbeitet wird und damit die Startadresse des Bootloaders sowie die Fuses des ATmega richtig eingestellt werden. Die eingeschlagene Lösung erzeugt während der Abarbeitung der Einzelschritte für die Erzeugung des Programmcodes für den Bootloader noch weitere kleine Dateien, welche in den nachfolgenden Schritten für die Anpassung der Start-Adresse und der Fuses erforderlich sind. Die Startadresse für den jeweiligen Zielprozessor ist abhängig von der vorhandenen Flash-Speichergröße, dem Speicherbedarf für den aktuellen Bootloader-Code und der Kachelgröße, die für Bootloader beim Zielprozessor zur Verfügung steht. Als Kachelgröße bezeichne ich die kleinste Speichergröße für Bootloader, die der jeweilige Prozessor zur Verfügung stellen kann.

Bei Prozessoren wie der ATtiny84, die keine Bootloader Startadresse einstellen können, wird die Seitengröße des Flash-Speichers für die Berechnung benutzt. Beim ATtiny84 sind das 64 Bytes. Damit liegt die Startadresse des Bootloaders immer am Anfang einer Flash Speicherseite.

Bei allen anderen Zielprozessoren kann der Bootloader-Bereich mit den Fuse-Bits BOOTSZ1 und BOOTSZ0 eingestellt werden (jeweils mit den Werten 0 und 1). Wenn man die beiden Bits zusammensetzt, ergibt sich daraus die Bootloader-Größe BOOTSZ mit Werten zwischen 0 und 3. Dabei bedeutet 3 immer den kleinsten mögliche Bootloader Speicherbereich. Der Wert 2 gibt den doppelten, der Wert 1 den vierfachen und der Wert 0 den acht-fachen Speicherbereich vor. Die Tabelle 2.2 auf Seite 10 gibt einen Überblick für verschiedene Zielprozessoren.

3.2 Automatische Größenanpassung in der optiboot Makefile

Die Bootloader Startadresse und die benötigte Bootloadergröße wird automatisch in der Makefile angepasst. Für die Berechnungen werden einige Zwischendateien erzeugt, was nur zusammen mit den folgenden Linux Werkzeugen funktioniert:

bc ein einfacher Rechner, der die Eingabe- und Ausgabe-Werte sowohl dezimal als auch sedezial (hex) verarbeiten kann.

cat gibt den Inhalt von Dateien auf der Standard-Ausgabe aus.

cut kann Teile von Zeilen einer Textdatei ausschneiden.

echo gibt den angegebenen Text auf der Standard-Ausgabe aus.

grep gibt nur Zeilen einer Textdatei mit dem angegebenen Suchtext aus.

tr kann Text-Zeichen ersetzen oder löschen.

Bisher ist die Funktion der Makefile nur mit einem Linux-System getestet. Wahrscheinlich ist die Benutzung unter Windows nur möglich, wenn das Cygwin Paket installiert wird.

Um die erzeugten Zwischendateien braucht man sich im Regelfall nicht zu kümmern. Hier möchte ich aber wenigstens die Namen und die Bedeutung erwähnen:

BootPages.dat enthält die Zahl der vom Bootloader benötigten Seiten. Bei Prozessoren mit Bootloader Unterstützung kann die Zahl nur 1, 2, 4 oder 8 sein und gibt an, das wie-vielfache der Mindest-Bootloadergröße verwendet wird. Bei der virtuellen Bootloader Seite kann die Zahl beliebig sein und gibt die Zahl der benötigten Flash-Speicherseiten an.

BOOTSZ.dat enthält eine Zahl zwischen 0 und 3 für die Einstellung der BOOTSZ0 und BOOTSZ1 Bits.

BL_StartAdr.dat enthält die Startadresse des Bootloaders im sedezial (hex) Format. Die Startadresse wird berechnet aus der Flash-Speichergröße des jeweiligen Zielprozessors und aus der Zahl der benötigten Speicherseiten.

EFUSE.dat enthält im sedezial Format der Wert für die efuse. Die Makefile entscheidet abhängig vom Zielprozessor, ob diese Datei verwendet wird.

HFUSE.dat enthält im sedezial Format der Wert für die hfuse. Die Makefile entscheidet abhängig vom Zielprozessor, ob diese Datei verwendet wird.

3.3 Zielvorgaben für optiboot Makefile

Die Steuerung des Ablaufs für die Generierung der Programmdateien aus dem Quellcode ist in der Makefile festgelegt. Außer der Haupt Makefile gibt es noch weitere drei Erweiterungen der Makefile, die automatisch von der Haupt Makefile integriert werden: Makefile.1284, Makefile.atmel, und Makefile.extras . Dabei können verschiedene Konfigurationen auch für einen Prozessortyp festgelegt worden sein. In der Tabelle 3.1 sind die derzeit vordefinierten Konfigurationen für AVR Prozessoren angegeben. Im Prinzip ist diese Liste natürlich erweiterbar. Die einstellbaren Parameter sind aber auch in der Aufrufzeile des make Programms als Parameter oder auch als Umgebungsvariable der Shell einstellbar.

Name	MCU	AVR_ FREQ	total Flash size	Flash page size	BP_ LEN	LFUSE	HFUSE	EFUSE
attiny84	t84	16M?	8K	64	(64)	62	DF	FE
atmega8	m8	16M	8K	64	256	BF	CC	-
atmega88	m88	16M	8K	64	256	FF	DD	04
atmega16	m16	16M	16K	128	256	FF	9C	-
atmega168	m168	16M	16K	128	256	FC	DD	04
atmega168p	m168p	16M	16K	128	256	FC	DD	04
atmega32	m32	16M	16K	128	256	BF	CE	-
atmega328	m328	16M	32K	128	512	FF	DE	05
atmega328p	m328p	16M	32K	128	512	FF	DE	05
atmega644p	m644p	16M	64K	256	512	F7	DE	05
atmega1284	m1284	16M	128K	256	512	F7	DE	05
atmega1284p	m1284p	16M	128K	256	512	F7	DE	05
atmega1280	m1280	16M	128K	256	1K	FF	DE	05

Tabelle 3.1. Prozessor targets für optiboot Makefile

Alle Angaben für Größen sind in Bytes angegeben, die Werte für die Fuses sind die sedezimalen Werte (hex). Die Frequenz-werte werden in Hz angegeben, 16M entspricht also 16000000 Hz. Die Standard Baud-Rate für die serielle Schnittstelle beträgt immer 115200.

Neben den universellen Konfigurationen gibt es auch Konfigurationen für bestimmte Platinen oder Arbeitsumgebungen. Die Tabelle 3.2 zeigt die unterschiedlichen Einstellungen.

Name	MCU	AVR_ FREQ	BP_ LEN	L FUSE	H FUSE	E FUSE	BAUD_ RATE	LED	SOFT_ UART
luminet	t84	1M	64v	F7	DD	04	9600	0x	-
virboot8	m8	16M	64v						
diecimila	m168	(16M)		F7	DD	04		3x	-
lilypad	m168	8M		E2	DD	04	-	3x	-
pro8	m168	16M		F7	C6	04	-	3x	-
pro16	m168	16M		F7	DD	04	-	3x	-
pro20	m168	16M		F7	DC	04	-	3x	-
atmega168p_lp	m168	16M		FF	DD	04	-		-
xplained168pb	m168	(16M)					57600		
virboot328	m328p	16M	128v						-
atmega328_pro8	m328p	8M		FF	DE	05	-	3x	-
xplained328pb	m168	(16M)					57600		
xplained328p	m168	(16M)					57600		
wildfire	m1284p	16M					-	3xB5	
mega1280	m1280	16M		FF	DE	05	-		-

Tabelle 3.2. vorkonfigurierte targets für optiboot Makefile

3.4 Die Optionen für die optiboot Makefile

Mit den Optionen wird die Eigenschaft des optiboot Bootloaders eingestellt. Beispielsweise kann mit der Option `SOFT_UART` veranlasst werden, daß ein Softwareprogramm für die serielle Kommunikation verwendet werden soll. Sonst wird, wenn vorhanden, die auf dem Chip integrierte serielle Schnittstelle mit den Pins TX (Transmit = senden) und RX (Receive = empfangen) benutzt. Bei mehreren integrierten seriellen Schnittstellen wird normalerweise die erste Schnittstelle mit der Nummer 0 verwendet. Es kann aber auch jede andere vorhandene Schnittstelle mit der Option `UART` vorgegeben werden (`UART=1` für die zweite Schnittstelle). Bei der Hardware UART Schnittstelle sind die Pins für Empfangen (RX) und Senden (TX) fest verknüpft. Bei der Software-Lösung für die serielle Schnittstelle sind alle Pins des AVR Prozessors frei für die serielle Kommunikation wählbar. Die einzige Bedingung ist, daß die Pins für digitale Eingabe (RX) beziehungsweise Ausgabe (TX) geeignet sind. Näheres zu den möglichen Optionen findet man in der Übersicht 3.3 und 3.4

Name der Option	Beispiel	Funktion
<code>F_CPU</code>	<code>F_CPU=8000000</code>	Teilt dem Programm die Taktrate des Prozessors mit. Die Angabe erfolgt in Hz (Schwingungen pro Sekunde). Das Beispiel gibt eine Frequenz von 8 MHz an.
<code>BAUD_RATE</code>	<code>BAUD_RATE=9600</code>	Gibt die Baud-Rate für die serielle Kommunikation an. Es werden immer 8 Datenbits ohne Parity verwendet.
<code>SOFT_UART</code>	<code>SOFT_UART=1</code>	Wählt für die serielle Kommunikation eine Software-Lösung.
<code>UART_RX</code>	<code>UART_RX=D0</code>	Gibt den Port und die Bitnummer für die seriellen Empfangsdaten an. Das Beispiel nimmt Bit 0 des D Ports für den seriellen Eingang.
<code>UART_TX</code>	<code>UART_TX=D1</code>	Gibt den Port und die Bitnummer für die seriellen Sendedaten an. Das Beispiel nimmt Bit 1 des D Ports für den seriellen Ausgang.
<code>UART</code>	<code>UART=1</code>	Wählt für die serielle Schnittstelle des Chips. Eine Auswahl setzt das Vorhandensein mehrerer Schnittstellen voraus. Nur wirksam ohne die <code>SOFT_UART</code> Option.
<code>LED_START_FLASHES</code>	<code>LED_START_FLASHES=3</code>	Wählt für die Anzahl der Blink-Zyklen für die Kontroll-LED.
<code>LED</code>	<code>LED=B3</code>	Wählt das Port-Bit für die Kontroll-LED. Beim Beispiel würde eine an das Bit 3 des Port B angeschlossene LED blinken. Bei der <code>LED_START_FLASHES</code> Option blinkt die LED die angegebene Anzahl vor dem Kommunikations-Start. Mit der <code>LED_DATA_FLASH</code> Option leuchtet die LED auch während des Wartens auf Daten.
<code>LED_DATA_FLASH</code>	<code>LED_DATA_FLASH=1</code>	Die Kontroll-LED leuchtet während des Wartens auf Empfangsdaten der seriellen Kommunikation.

Tabelle 3.3. Wichtige Optionen für die optiboot Makefile

Weitere Optionen sind in der Tabelle 3.4 aufgezählt. Diese Optionen sind nur für Software-Untersuchungen und für Prozessoren ohne Bootloader-Bereich interessant.

Name der Option	Beispiel	Funktion
SUPPORT_EEPROM	SUPPORT_EEPROM=1	Wählt für das Bootloader-Programm die Lese- und Schreibfunktion für EEproms. Wenn als Quelle das Assembler-Programm gewählt wurde, ist die EEprom Unterstützung ohne gesetzte Option eingeschaltet, kann aber abgeschaltet werden, wenn die SUPPORT_EEPROM Option auf 0 gesetzt wird. Bei der C-Quelle muß die Funktion mit der Option eingeschaltet werden (Standard = aus).
C_SOURCE	C_SOURCE=1	Wählt als Programmquelle das C-Programm anstelle des Assembler-Programms (0 = Assembler). Die Assembler Version benötigt weniger Speicherplatz.
BIGBOOT	BIGBOOT=512	Wählt zusätzlichen Speicherverbrauch für das Bootloader-Programm. Das dient nur zum Test der automatischen Anpassung an die Programmgröße in der Makefile.
VIRTUAL_BOOT_PARTITION	VIRTUAL_BOOT_PARTITION	Ändert die Programmdatei eines Anwenderprogramms so ab, daß der Bootloader beim Reset angesprochen wird. Für den Start des Anwenderprogramms wird ein anderer Interrupt-Vektor benutzt.
save_vect_num	save_vect_num=4	Wählt eine Interrupt-Vektornummer für die VIRTUAL_BOOT_PARTITION Methode aus.

Tabelle 3.4. Weitere Optionen für die optiboot Makefile

3.5 Benutzung von optiboot ohne Bootloader-Bereich

Für Prozessoren ohne speziellen Bootloader-Bereich im Flash-Speicher wie dem ATtiny84 ist eine Möglichkeit vorgesehen, trotzdem optiboot zu benutzen. Diese Funktion wird mit der Option VIRTUAL_BOOT_PARTITION gewählt. Dabei wird im Anwenderprogramm auf der Reset-Vektoradresse die Start-Adresse des Bootloaders eingetragen damit bei einem Reset immer der Bootloader zuerst angesprochen wird. Die Start-Adresse des Anwender-Programms wird dabei auf die Adresse eines anderen Interrupt-Vektors verlegt. Dieser Interrupt-Vektor sollte vom Anwenderprogramm nicht benutzt werden. Wenn der Bootloader in angemessener Zeit keine Daten von der seriellen Schnittstelle empfängt, springt er zu dem Sprungbefehl, der auf der „Ersatz“-Vektoradresse steht und startet damit das Anwenderprogramm. Die Abbildung 3.1 soll die Veränderung verdeutlichen.

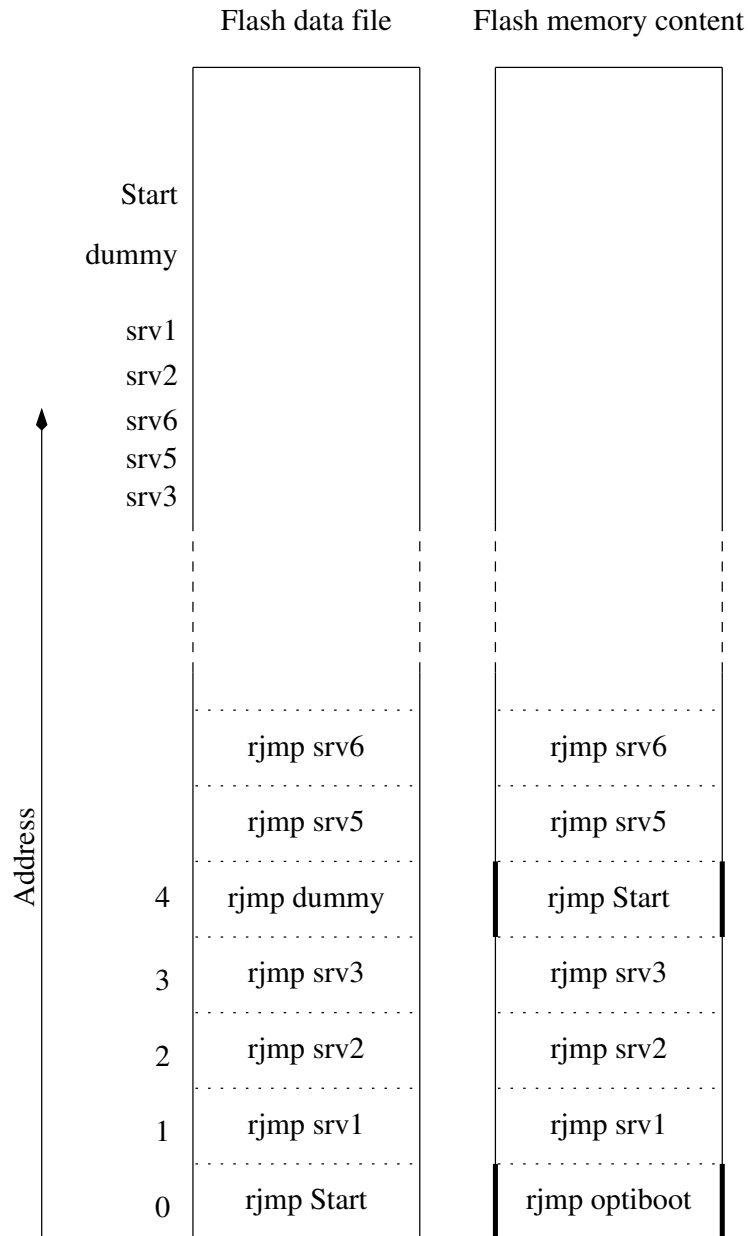


Abbildung 3.1. Veränderung der Programmdatei durch optiboot

Auf der linken Seite ist der Inhalt der Datei dargestellt, welche die Programmdatei (.hex) enthält. Rechts daneben ist der Inhalt des Flash-Speichers dargestellt, wie er vom Optiboot Bootloader geschrieben wird. An zwei Interruptvektor-Adressen wurde der Inhalt verändert. Einmal wurde auf dem Reset-Vektor 0 der Optiboot Bootloader eingetragen und zum anderen auf der „Ersatz“-Vektor Adresse 4 ein Sprung zum Start des Applikations-Programms eingetragen. Eine Schwierigkeit bei der Methode entsteht aber dadurch, daß die Programmdatei nach der Programmierung meistens zur Kontrolle zurückgelesen werden. Damit bei der Kontrolle keine Fehler gemeldet werden, gibt Optiboot nicht den wirklichen Inhalt der Interrupt-Vektortabelle zurück, sondern den Zustand vor seiner Manipulation. Die Sprung-Adresse im Reset-Vektor kann dafür aus den Daten des „Ersatz“-Interruptvektors rekonstruiert werden. Aber die ursprünglichen Daten des „Ersatz“-Interruptvektors wären verloren, da sie an keiner Stelle der Vektortabelle wiederzufinden sind. Optiboot benutzt zum Sichern des Original-Inhaltes des „Ersatz“-Vektors deshalb die beiden letzten Bytes des EEprom-Speichers. Damit ist eine Kontrolle der Programmdatei solange möglich, wie die beiden letzten Bytes des EEproms nicht überschrieben werden. Selbst wenn die EEprom Daten überschrieben werden, bleibt der Bootloader funktionsfähig. Nur die Kontrolle (verify) der Programmdatei ist dann

nicht mehr möglich. Bei der Adresse des „Ersatz“-Interruptvektors wird dann ein Fehler gemeldet.

Bei Prozessoren mit mehr als 8 kByte Flash Speicher werden zwei Befehls Worte für jeden Interrupt-Vektor vorgesehen. Normalerweise stehen auf diesen Doppelworten jmp Befehle mit den jeweiligen Sprungzielen. Auch diese Art der Vektortabelle wird von Optiboot berücksichtigt. Wenn aber für das Bindeprogramm (Linker avr-ld) die Option `-relax` verwendet wird, werden alle jmp Befehle durch die kürzeren rjmp Befehle ersetzt, wenn dies bei der jeweiligen Sprungadresse möglich ist. Dies wird derzeit nicht von optiboot berücksichtigt. Das Optiboot Programm geht fest davon aus, daß in der Vektortabelle jmp Befehle stehen, wenn mehr als 8 kByte Flash-Speicher vorhanden sind. Deshalb wird die `VIRTUAL_BOOT_PARTITION` Methode meistens nicht funktionieren, wenn die `-relax` Option beim Programmbinden benutzt wurde!

Weiter ist zu beachten, daß bei Benutzung der `VIRTUAL_BOOT_PARTITION` Option für Prozessoren, die auch die normale Bootloader Unterstützung bieten, das `BOOTRST` Fuse Bit nicht aktiviert wird. Der Grund hierfür ist, daß bei Benutzung der `VIRTUAL_BOOT_PARTITION` die Start-Adresse des Bootloaders auf einer anderen Adresse liegen kann wie bei der normalen Bootloader Unterstützung. Bei Benutzung der Option `VIRTUAL_BOOT_PARTITION` kann die Startadresse auf jedem Anfang einer Seite des Flash-Speichers liegen. Bei der normalen Bootloader Unterstützung kann immer nur das einfache, doppelte, vierfache oder achtfache einer Mindest-Bootloadergröße berücksichtigt werden (`BOOT_SZ` Fuse-Bits), wie es in Abbildung 2.3 auf Seite 9 dargestellt wird.

Literaturverzeichnis

- [1] Atmel Corporation *8-bit AVR with 8KBytes In-System Programmable Flash - ATmega8(L)*,. Manual, 2486AA-AVR-02/13, 2013
- [2] Atmel Corporation *8-bit AVR with 16KBytes In-System Programmable Flash - ATmega16A*,. Manual, 8154C-AVR-07/14, 2014
- [3] Atmel Corporation *8-bit AVR with 32KBytes In-System Programmable Flash - ATmega32(L)*,. Manual, doc2503-AVR-07/11, 2011
- [4] Atmel Corporation *8-bit AVR with 4/8/16/32KBytes In-System Programmable Flash - ATmega48 - ATmega328*,. Manual, 8271J-AVR-11/15, 2015
- [5] Atmel Corporation *8-bit AVR with 16/32/64/128KBytes In-System Programmable Flash - ATmega164 - ATmega1284*,. Manual, 8272G-AVR-01/15, 2015
- [6] Atmel Corporation *8-bit AVR with 2/4/8KBytes In-System Programmable Flash - ATtiny24-ATtiny44-ATtiny84* ,. Manual, doc8006-AVR-10/10, 2010
- [7] Atmel Corporation *STK500 Communication Protocol* ,. Application Note, AVR061-04/03, 2003
- [8] http://en.wikibooks.org/wiki/LaTeX/LaTeX_documentation,. Guide to the LaTeX markup language, 2012
- [9] <http://www.xfig.org/userman> *Xfig documentation*,. Documentation of the interactive drawing tool xfig, 2009
- [10] <http://www.cs.ou.edu/~fagg/classes/general/atmel/avrdude.pdf> *AVRDUDE Userguide*,. A program for download/uploading AVR microcontroller flash and eeprom, by Brian S. Dean 2006
- [11] <http://www.mikrocontroller.net/articles/AVRDUDE> *Online Dokumentation des avrdude programmer interface*, Online Article, 2004-2011