

AVR Assembler für komplexe Projekte

Autor: Dr. Thomas Redelberger redetho@gmx.de

Version 0.3 2017-11-26

Inhalt

1.	Rechtliches	2
	Copyright	2
	Haftungsausschluss	2
	Übersetzungen	2
2.	Vorwort	2
3.	AVRASM2 Haupt-Eigenschaften.....	3
	Kein Linking	3
	AVRASM2 verarbeitet den Source-Code zweimal	3
	C-Style Pre-Processor	3
	Dynamisch erzeugte Symbole.....	3
4.	Prinzipien	3
	Den Source-Code wohl strukturieren	3
	Code soweit wie möglich portabel halten	4
	Hohe Geschwindigkeit anstreben.....	4
5.	Wie man den Source-Code strukturieren kann	4
	Wie das Dateisystem helfen kann, Code zu strukturieren	4
	Unterschied zwischen Declaration-Code und Definition-Code	5
	Lexikalische Reihenfolge der Include-Dateien	5
	Programm-Bausteine.....	6
6.	Fast Alles dreht sich um Assembler-Symbole.....	6
	Gültigkeitsbereich von Namen.....	7
	Namensräume	7
	Labels für die Steuerung des Programmablaufs	8
7.	Funktionen und Interrupt-Handler bauen	9
	Wie man Makros nutzt, um Geschwindigkeit gegen Platz einzutauschen	9
	Makros in Include-Dateien.....	10
	Die Sichtbarkeit von Funktionen und Daten einschränken	10
	Funktionen, die nur lokal in einem Makro bekannt sind	10
	Variablen, die nur lokal in einem Makro bekannt sind.....	11
8.	Wozu kann man den C-artigen Pre-Processor nutzen?.....	12
9.	Lexikalische Details	13
10.	Prozessor-Ressourcen optimal nutzen	13
	Wann den Stack nutzen?	13
	Auf Strukturen und Arrays zugreifen	13
	Structs als Teil statischer Objekte	15
	Sonderfall EEPROM.....	16
	Welche Register für was verwenden?	16
11.	Program-Ablauf und Gleichzeitigkeit	16
	Hardware arbeitet oft parallel	16
	Wie mehrere Tasks quasi gleichzeitig ablaufen können	17
	Programmfluss-Beispiel.....	17

12. Tipps und Tricks	18
Wie man Code kürzer und lesbarer machen kann	18
Portierbarkeit	18
13. Beispiel für eine Entwicklung-Umgebung unter Windows	19
14. Referenzen	19
15. Anhang 1 - Unzulänglichkeiten des C-Pre-Processors	19
16. Anhang 2 - Konvention, wie Register verwendet werden	20
17. Anhang 3 - Makros, um mit Strukturen umzugehen	21
18. Anhang 4 - Makro, um einen neue FIFO-Instanz anzulegen	21
19. Anhang 5 - Beispiel-Projekt: Software für eine Synthesizer-Bedienoberfläche	22

1. Rechtliches

Copyright

Copyright © 2016-2017 Dr. Thomas Redelberger redetho@gmx.de. Alle Rechte liegen beim Autor.

Sie können dieses Dokument zitieren und kopieren, wenn Sie auf die originale Web-Seite verweisen und mich als Urheber angeben. Dieser Abschnitt "Rechtliches" muss Bestandteil sein.

Haftungsausschluss

Dieses Dokument dient nur der Information. Ich übernehme keinerlei Gewährleistung oder Garantie für Irgendetwas im Zusammenhang mit dem Inhalt dieses Dokuments.

Übersetzungen

Die erste Version dieses Dokuments hatte ich in Englisch verfasst und sie erschien 2017-09-15. Diese deutsche Version ist von mir, das ist meine Muttersprache. Ich ermutige jeden, der in eine andere Sprache übersetzen möchte, vorausgesetzt, er verweist auf dieses Original und mich.

2. Vorwort

Dieses Dokument richtet sich an Alle, die Assembler-Code schreiben wollen, der gut strukturiert, wiederverwendbar, gut wartbar und effizient ist. Es basiert auf meinen eigenen Erfahrungen mit Microchip AVR 8-Bit-Mikrocontrollern, die ich in Assembler programmiere. Es ist aber *keine* Einführung in die Programmierung der AVR-Mikrocontroller. Ich behandle hier weder die AVR 8-Bit Maschinen-Instruktionen, noch wie man bestimmte Algorithmen in Assembler umsetzt. Dafür verweise ich auf die originale Dokumentation von Microchip [1](#)) und viele gute Web-Seiten im Netz [2](#)).

Ich hatte mich entschieden, meine ersten Schritte mit AVR-Mikrocontrollern in Assembler zu machen, um von Grund auf zu lernen, wie man sie nutzt. Bei meinem ersten größeren AVR ATmega-Projekt blieb ich bei Assembler, weil das Projekt sehr hardwarenah war, mit viel Interfacing und sehr wenigen mathematischen Berechnungen.

Ich nutze den AVR-Assembler von Microchip (vormals Atmel). Viele Tipps können aber auch mit anderen Assemblern verwendet werden. Mit AVR-Assembler meine ich hier den AVR Assembler 2, abgekürzt AVRASM2. Der ältere AVRASM, der früher mit AVR-Studio mitgeliefert wurde, unterstützt viele der hier vorgestellten Funktionen nicht.

Für viele Projekte wird Assembler nicht das beste Werkzeug sein. Mit GCC aus der AVR-GNU-Toolchain steht ein ausgereifter C-Compiler zur Verfügung. Viel Beispiel-Code und Libraries sind gewichtige Vorteile. Man sollte auch den GNU Assembler (GNU AS oder GAS abgekürzt; in der AVR-Welt heißt er "avr-as") in Betracht ziehen. Er braucht vermutlich mehr Einarbeitungszeit als AVRASM2, ist aber sehr leistungsfähig und kann vorteilhaft mit GCC kombiniert werden.

3. AVRASM2 Haupt-Eigenschaften

Kein Linking

AVRASM2 erzeugt aus dem Source-Code direkt ausführbaren Code, genauer gesagt eine HEX-Datei-Darstellung von ausführbarem Code. Es gibt keine Zwischenschritte, es gibt keine Object- oder Library-Dateien. Das hat die folgenden Vor- und Nachteile:

- Man nutzt immer die aktuelle Code-Version. Weil man immer "Alles" assembliert, braucht man nicht zu prüfen, welche Codeteile geändert wurden, was man sonst mit einem Werkzeug wie "make" durchführt
- Abhängigkeiten sind explizit im Source-Code festgelegt. Demgegenüber braucht ein Link-Schritt immer eine Überprüfung, was von was abhängt und was neu assembliert werden muss. Auch das wird üblicherweise mit "make" automatisiert.
- Der Assembler braucht umso länger, je komplexer und länger das Projekt ist, weil ja immer Alles assembliert wird. Allerdings ist das auf heutiger PC-Hardware kaum ein Problem. Beispiel: eines meiner Projekte bestehend aus 29 Dateien und insgesamt 3 000 Lines of Code (LOC) assembliert in weniger als einer Sekunde.
- Es gibt nur eine Assembly-Unit und damit nur einen globalen Namensraum. Das erfordert besondere Sorgfalt, um Namenskollisionen zu vermeiden
- Code wieder zu verwenden ist schwieriger als zum Beispiel in einer GCC/avr-as-Umgebung. Man muss sich um Alles selbst kümmern. Code Anderer zu verwenden, ist kaum möglich, weil es keine vorgeschlagenen, geschweige denn verbindlichen, Standards für Software-Schnittstellen gibt
- Assembler-Code und Code in anderen Programmiersprachen können nicht kombiniert werden. Ich halte das für die schwerwiegendste Einschränkung, denn bei großen und komplexen Projekten möchte man eine jeweils angemessene Sprache verwenden.

AVRASM2 verarbeitet den Source-Code zweimal

AVRASM2 liest und verarbeitet den Source-Code zweimal. Im ersten Durchlauf (Pass) füllt er verschiedene interne Tabellen, auf die er im zweiten Durchlauf zurückgreifen kann. Das hat den Vorteil, dass man sehr große Freiheit hat, wie man den Source-Code anordnet.

Zum Vergleich: viele C-Compiler durchlaufen den Code nur einmal. Das hat zum Beispiel die Konsequenz, dass Include-Dateien mit Deklarationen (siehe unten) am Anfang der Source-Codes eingelesen werden müssen.

C-Style Pre-Processor

Atmel hat AVRASM2 mit einem Pre-Processor ausgerüstet, der wie der Pre-Processor arbeitet, den übliche C-Compiler haben. Die Pre-Processor-Anweisungen beginnen dem "#"-Zeichen (Doppelkreuz) statt "." (Punkt) wie die Assembler-Anweisungen. Der AVRASM2 Pre-Processor soll laut Microchip-Dokumentation wie ein C-Pre-Processor arbeiten.

Dynamisch erzeugte Symbole

Auf den ersten Blick bietet AVRASM2 keine Möglichkeit, Symbole dynamisch zu erzeugen. Mittels geschickter Makro-Programmierung lässt sich auch das erzielen. Damit kann man zum Beispiel Kontrollstrukturen bauen, wie das C-Konstrukt `switch`.

4. Prinzipien

Wie bei anderen Programmiersprachen, gibt es einige Prinzipien, die man auch bei Assembler beherrsigen sollte:

Den Source-Code wohl strukturieren

Keine Frage, Assembler-Source-Code kann schlimm ausschauen. Solcher Code ist schwer zu verstehen, schwer zu debuggen und schwer zu warten. Deshalb sollte man

- eine klare Code-Hierarchie anlegen, die hilft die Code-Struktur zu verstehen und den Code-Zweck
- den Code modularisieren, damit man möglichst viel wiederverwenden kann.

Code soweit wie möglich portabel halten

Mit portablem Code ist hier gemeint, dass der Code auf hinreichend ähnlichen AVR-Controllern mit möglichst wenigen Änderungen assembliert werden kann.

Hohe Geschwindigkeit anstreben

- Assembler-Code kann schneller sein als Code, der aus einer Hochsprache resultiert. Aber das sollte man nicht überschätzen, denn moderne Hochsprachen-Compiler kommen sehr nah an optimalen Code heran
- Was man zur Entwicklungszeit (Assembler-Zeit) erzielen kann, sollte man auch zur Entwicklungszeit machen, statt zur Laufzeit. Assembler hat Eigenschaften und Mittel genau das zu leisten, ohne dass die Lesbarkeit oder die Wartbarkeit des Source-Codes leidet.

All das verlangt nach den fortgeschrittenen Mitteln von AVRASM2. Diese Mittel haben manchmal allerdings den Nachteil, dass die Code-Portierung auf einen anderen Assembler aufwendiger wird, zum Beispiel nach avr-as.

5. Wie man den Source-Code strukturieren kann

Wie das Dateisystem helfen kann, Code zu strukturieren

Klar ist, dass man die Komplexität eines Problems dadurch reduzieren kann, indem man es in kleinere, weniger komplexe Teile teilt; Stichwort: divide and conquer. Beim Programmieren heißt das, den Source-Code in kleine Teile zu zerlegen. Auch AVRASM2 erlaubt, den Source-Code auf mehrere Dateien aufzuteilen, unter Nutzung des normalen Dateisystems des Betriebssystems (also Windows oder Linux, auf dem AVRASM2 mittels "Wine" läuft), wie bei den meisten Entwicklungs-Werkzeugen.

Die AVRASM2 `.include`-Anweisung erlaubt den Source-Code baumartig zu strukturieren. Ein Projekt hat dabei eine Wurzel-Datei, die weitere Source-Code-Dateien per `.include` einbindet. Diese Dateien wiederum können andere Dateien einbinden und so weiter, was von der Struktur her einem Baum gleicht. Wenn ein Include-File ein weiteres Include enthält, ist ein relativer Pfadname übrigens *nicht* relativ zum Include-File, sondern zu dem ursprünglichen Quell-Text, der das erste Include enthält.

Zu beachten ist, dass der Code, der vom Assembler lexikalisch zuerst "gesehen" wird, auch den Anfang des ausführbaren Codes bildet. Dieser wird beim Einschalten der Mikrocontroller-Spannungsvorsorgung oder beim Reset zuerst ausgeführt.

AVRASM2 hat keine Einschränkungen bei Include-Dateien und lässt den Programmierenden alle Freiheiten, was er in Include-Dateien schreibt. Die Include-Dateien werden einfach eins-zu-eins am Punkt der `.include`-Anweisung eingefügt. Also hat der Programmierer auch volle Freiheit, wie er den Code strukturiert.

Eine plausible Möglichkeit Include-Datei zu bilden ist, allen Code, der zu einem Objekt gehört, in eine Include-Datei zu schreiben. Passende Objekte zu finden, ist in Assembler nicht anders als in anderen Programmiersprachen. Zum Beispiel können Hardware-Einheiten als Objekte angesehen werden. Zum Beispiel ordne ich allen Code, der zur UART-Bedienung gehört, in eine Include-Datei, den Code, der zu TWI gehört, in eine Andere. Die Idee dahinter ist natürlich, die Include-Dateien und damit die "Objekte" in anderen Projekten wiederverwenden zu können.

Es gibt verschiedene Möglichkeiten zu überprüfen, ob die Wahl der Objekte vernünftig ist. Eine ist zum Beispiel, ob die Include-Dateien grob ähnliche Größe haben, wobei man annimmt, dass die Größe mit der Komplexität korreliert. Um im Beispiel des Baumes zu bleiben: man überprüft, ob die Äste ähnlich dick sind, damit der Baum wohlgeformt aussieht.

Den Code als Baum zu strukturieren, ist in AVRASM2 eine Notwendigkeit, weil es kein anderes Mittel gibt und keinen "Link-Step". Linking arbeitet ja auch in einer baumartigen Weise, weil Code, der aus Object-Dateien oder Libraries hinzugelinkt wird, seinerseits weiteren Code "nachziehen" kann und so weiter.

Unterschied zwischen Declaration-Code und Definition-Code

Ein wesentlicher Gesichtspunkt einer Programmiersprache ist, welche Sprachkonstrukte tatsächlich ausführbaren Code erzeugen - ich nenne diese Sprachkonstrukte "Definitionen" - und welche Sprachkonstrukte nur Hilfsmittel sind, die keinen ausführbaren Code erzeugen - ich nenne diese "Deklarationen". Bei AVR-Mikrocontrollern bedeutet ausführbarer Code, Code, der im Programmspeicher-Segment (`.cseg`) angelegt wird oder Daten im EEPROM-Segment (`.eseg`). Eine Sonderstellung nehmen Sprachkonstrukte ein, die sich auf das RAM-Segment (`.dseg`) auswirken. Sie erzeugen zwar keinen ausführbaren Code, sondern reservieren nur Speicherplatz. Ich zähle sie aber trotzdem zu den Definitionen.

Der Hauptunterschied zwischen Assembler und Hochsprachen ist, dass der Programmierer die volle Kontrolle hat, welcher Code erzeugt wird. Daher ist die Unterscheidung zwischen Definitionen und Deklarationen sehr wichtig, aber nicht offensichtlich:

- Assembler-Anweisungen (directives) deklarieren Dinge, zum Beispiel symbolische Namen für Konstanten. Dabei sind `.db`, `.dw`, `.dd`, und `.dw` Ausnahmen. Diese Anweisungen erzeugen Code. Die Anweisungen `.byte` und `.org` sind Spezialfälle, die zwar keinen Code erzeugen, aber Speicherplatz reservieren
- Assembler-Instruktionen (instructions), gemeint sind Maschineninstruktionen (processor instructions), erzeugen Maschinen-Code und definieren so den Programmablauf. Ausnahmen sind Instruktionen, die in einer Makro-Deklaration enthalten sind, das heißt zwischen den Anweisungen `.macro` and `.endm` stehen. Diese Instruktionen werden vom Assembler temporär gespeichert und erzeugen Code erst beim Makro-Aufruf.
Man kann sich Makro-Deklarationen wie kleine interne Include-Dateien vorstellen, die der Assembler im RAM des Entwicklungs-PCs ablegt. Beim Makro-Aufruf fügt der Assembler den Code-Abschnitt dann eins-zu-eins ein, analog einem `.include`.
Eins-zu-eins stimmt nicht ganz, denn man kann den Code mit Hilfe von Makro-Argumenten zum Aufruf-Zeitpunkt verändern.
- Um den Code weiter zu strukturieren, nutze ich ein Verfahren, das auch in anderen Sprachen gebräuchlich ist, zum Beispiel in C. Ich unterscheide zwischen Include-Dateien, die nur Deklarationen enthalten und daher zunächst keinen Code erzeugen, wenn sie mittels `.include` eingebunden werden. Meine Konvention ist, diesen die Datei-Endung `.inc` zu geben.
Demgegenüber haben Include-Dateien die Endung `.asm`, wenn sie Definitionen enthalten und deshalb Code erzeugen.

Lexikalische Reihenfolge der Include-Dateien

Die lexikalische Reihenfolge ist die Reihenfolge, in der der Assembler den Source-Code "sieht", nachdem alle Include-Dateien eingefügt worden sind und der Source-Code als eine einzige, lange Abfolge von Anweisungen verarbeitet wird.

Weil AVRASM2 den Source-Code zweimal verarbeitet (two-pass assembler), hat man sehr viel Freiheit, in welcher Reihenfolge man was anordnet. Wie könnte eine leicht nachvollziehbare Reihenfolge aussehen?

Ich verlasse mich dazu nicht allzu sehr auf die "two-pass"-Eigenschaft von AVRASM2, sondern orientiere mich daran, wie man üblicherweise C-Code anordnet. Dort gibt es strengere Vorgaben, weil die meisten C-Compiler den Code nur einmal durchlaufen:

- Alle Deklarationen, die der einbindende Teil des Source-Codes verwendet, werden per `.include` eingebunden *bevor* sie verwendet werden. In anderen Worten: die Include-Dateien, die Deklarationen enthalten, werden an der Spitze des einbinden Source-Codes eingebunden. Solchen Include-Dateien gebe ich die Endung `.inc`. Sie entsprechen `.h`-Dateien in C
- Include-Dateien, die Definitionen enthalten, d.h. die Code erzeugen, kommen in Dateien der Endung `.asm`. Solche binde ich in der Regel am Ende der Wurzel-Source-Code-Datei ein. Dies ersetzt die fehlende Linking-Funktionalität.

Ein Beispiel, wie ich UART-Code auf diese Weise wiederverwende:

```
...
.include "trUARTS.inc"
...
rjmp      RESET          ; Interrupt vectors follow
```

```

.org      URXC0addr          ; USART0 receive complete
rjmp     trUARTrxInterrupt0
.org      UDRE0addr         ; USART0 transmitter register empty
rjmp     trUARTtxInterrupt0
...
.set      TRUART_RX_BUF_SIZE0 = 256      ; default in trUARTS.inc is 16
MtrUARTinit 0, BAUD_MIDI
...
;; send a byte
rcall    trUARTtx0
...
...

;;; lowest level UART access; this also includes the interrupt handlers
.include "trUARTS.asm"

```

Das Symbol `TRUART_RX_BUF_SIZE0` ist in der Datei `trUARTS.inc` festgelegt auf den Default-Wert 16. Der einbindende Code ändert das auf 256, um einen längeren UART-Empfangspuffer zu bekommen. Darauf aufbauend wird in der Datei `trUARTS.asm` 256 Bytes RAM (`.dseg`) reserviert, um diesen Puffer bereitzustellen.

Das Assembler-Makro `MtrUARTinit` ist in der Datei `trUARTS.inc` definiert. Es gibt keine Funktion `trUARTinit`, weil die Hardware-Initialisierung in den allermeisten Projekten lexikalisch nur einmal vorkommt. Deshalb ist eine Funktion überflüssig, leistet nicht mehr, kostet aber ein paar Byte mehr Programmspeicher. `trUARTtx0` dagegen ist eine Funktion, die in `trUARTS.asm` definiert ist. Es gibt ein zugehöriges Assembler-Makro in `trUARTS.inc`. Weiter unten werde ich den Grund dazu erläutern. Interrupt-Handler wie `trUARTrxInterrupt0` sind in `trUARTS.asm` definiert.

Programm-Bausteine

Die kleinsten Bausteine der Programm-Funktionalität sind traditionell Funktionen, Sub-Routinen oder Methoden-Aufrufe.

Assembler bietet diese natürlich auch. Der Programmierer hat nur mehr Arbeit damit als in Hochsprachen. Mit Makros bieten Assembler aber noch ein weiteres Sprachmittel, das in Hochsprachen oft nicht verfügbar ist (die C-Pre-Processor-Makros sind nicht ganz so leistungsfähig wie ihre Assembler-Verwandten).

Assembler-Makros können zum Beispiel zum "Inlining" von Code verwendet werden. Der Programmierer legt das selbst fest. In Hochsprachen entscheidet der Compiler, welche Funktionen "inlined" werden. Der Programmierer kann das zu einem gewissen Grade beeinflussen, zum Beispiel über Compiler-Optionen.

Ich beschreibe weiter unten, wie man die Wahl zwischen Funktion und Inline-Code mit AVRASM2 effizient gestaltet.

6. Fast Alles dreht sich um Assembler-Symbole

Symbole sind ein Assembler-Kern-Konzept. In AVRASM2 stehen Symbole immer für Zahlen. Wenn man Strings symbolische Namen geben will, muss man den C-artigen Pre-Processor verwenden. Symbole können für beliebige konstante Zahlen stehen, aber auch für Adressen im Programmspeicher (`.cseg`), im EEPROM (`.eseg`) oder RAM (`.dseg`).

Labels sind spezielle Symbole, die immer für Speicheradressen stehen. Syntaktisch stehen sie am Anfang der Zeile und werden mit einem Doppelpunkt abgeschlossen. Ein Label, der für eine Adresse im Programmspeicher (`.cseg`) steht, bezeichnet also entweder ein Sprungziel oder eine Einsprung-Adresse für eine Funktion oder einen Interrupt-Handler. Labels, die für Adressen im EEPROM (`.eseg`) oder RAM (`.dseg`) stehen, werden für Speicheroperationen genutzt. Durch Labels braucht man nie absolute Speicheradressen anzugeben.

Wie erwähnt sind Labels auch nur Symbole. Zwar führt AVRASM2 Buch und merkt sich, wie Symbole und Labels deklariert wurden, wie man in der Map-Datei sehen kann, die AVRASM2 optional erzeugt. (Standard-Dateiendung: `map`). Allerdings gibt es keinerlei Einschränkungen wie Labels und Symbole verwendet werden können und bei Verwechslungen keine Assembler-Warnungen.

Ich habe deshalb die Konvention übernommen, Symbole, die für Konstanten stehen, mit Großbuchstaben zu schreiben und Unterstrichen. Symbole, die gemischt groß/klein geschrieben sind und klein beginnen, stehen für Adressen. Bei Konstanten folge ich einer weiteren Konvention, den Zweck der Konstanten anzuzeigen. Dazu hänge ich einen Suffix an:

XYZ_OFFS	Offset, z.B. Byte-Offset von einem Element einer Struktur zum Anfang der Struktur
XYZ_BITNO	Bit-Nummer, von 0-7
XYZ_MASK	Bit-Maske, z.B. 0x0F
XYZ_LEN	Länge, z.B. eines Arrays oder einer Struktur
XYZ_SIZE	Das gleiche wie LEN
XYZ_NUM	Anzahl, z.B. Anzahl Array-Elemente
XYZ_END	Index des letzten Array Elements = höchster Array-Index d.h. XYZ_END = XYZ_LEN - 1, weil der Index mit 0 beginnt

Die Suffixe sagen allerdings nichts über den Datentyp aus. Für AVRASM2 sind alle Symbole 64-Bit Integers.

Gültigkeitsbereich von Namen

Eine der Haupt-Eigenschaften von AVRASM2 ist der direkte Weg vom Source-Code zum ausführbaren Code. Jedes Projekt besteht nur aus einer einzigen "Unit", nämlich dem Wurzel-Source-Code und allen eingebunden Include-Dateien. Für den Assembler ist das alles eine einzige, lange Abfolge von Anweisungen.

Das hat zu Folge, dass alle Symbole global sind. Um Kollisionen (clash) zu vermeiden, muss man ein System einführen, das sicherstellt, dass Symbolnamen eindeutig sind. Auch das muss die Namenskonvention leisten. Ich verwende Präfixe, um Namens-Kollisionen zu vermeiden. Jeder Symbolname beginnt mit einer kurzen Buchstabenfolge, die aus dem Dateinamen der Include-Datei abgeleitet ist, in der das Symbol deklariert wird. Damit vermeidet man auch Kollisionen mit den zahlreichen Symbolen für Hardware-Adressen und Bit-Nummern, die in den offiziellen Include-Dateien von Microchip deklariert werden.

Beispiel: es gibt die Dateien trUARTS.inc and trUARTS.asm mit UART- Code. Da gibt es zum Beispiel das Symbol `trUARTTrxInterrupt0` für die Einsprung-Adresse zum Empfänger-Interrupt-Handler und `TRUART_RX_BUF_SIZE0`, welches die Größe des Empfänger-FIFO-Puffers bestimmt.

Es gibt viele Schreib-Konventionen und noch mehr Debatten darüber, welche besser sind. Wichtig ist, dass man sich für eine Konvention entscheidet und diese dann auch durchhält.

Zum Vergleich: avr-as braucht einen nachfolgenden Link-Schritt. Dadurch kann man ein Projekt in verschiedene Objekte (units, linking objects) aufteilen, die getrennt voneinander assembliert werden und hernach gelinkt. Die einzelnen Assemblierungs-Units haben dadurch getrennte Symbol-Namensräume und der Programmierer kann entscheiden, ob ein Symbol nur in einer Unit bekannt sein soll oder global im ganzen Projekt. Dem Linker werden globale Symbole bekannt gemacht und er kann sie zu Auflösung von Referenzen verwenden. Von lokalen Symbolen bekommt der Linker nichts mit - es sei denn, sie werden extra zu dem Zweck mitgegeben, später dem Debugger bekannt zu sein.

Namensräume

Assembler bieten nur wenige Namensräume im Vergleich zu Hochsprachen. In AVRASM2 können die folgenden Symbole und Namen erzeugt werden:

1. Symbole (mit `.equ` oder `.set` deklariert)
2. Symbolische Namen für Prozessor-Register (mit `.def` deklariert)
3. Makro-Namen (mit `.macro` deklariert)
4. CSEG-Labels
5. DSEG-Labels
6. ESEG-Labels.
7. CSEG-Labels in einer Makro-Deklaration
8. DSEG-Labels in einer Makro-Deklaration
9. ESEG-Labels in einer Makro-Deklaration

Allerdings stellt sich heraus, dass es für die Fälle 1. bis 6. pro Projekt nur *einen*, globalen Namensraum gibt. Das heißt zum Beispiel, dass ein Makro nicht so heißen darf, wie ein Label oder ein Symbol. Für die Fälle 7. bis 9. gibt es pro Makro-*Aufruf* je einen Namensraum. Alle Assembler können lokale Labels erzeugen, die nur lokal in einem Makro gelten. Bei manchen Assemblern muss man dazu eine spezielle Syntax verwenden, zum Beispiel bei avr-as. AVRASM2 hat die Eigenheit, dass *alle* Labels lokal sind, die in einem Makro deklariert werden. Weil es für jeden Makro-Aufruf einen separaten Label-Namensraum gibt, kollidieren Labels also nicht

- wenn dasselbe Makro mehrfach aufgerufen wird
- wenn andere Makros dieselben Label verwenden.

Weiter unten beschreibe ich, wie man mit solchen lokalen Labels arbeitet und dadurch auch die Einschränkungen abmildert, die daher rühren, dass man nur eine Assembly-Unit pro Projekt hat.

Gleichwohl *kann* man in einem Makro ein Symbol erzeugen, das man wie einen globalen Label verwenden kann, mit der Syntax:

```
.equ      myGlobalLabel = PC
```

Dies scheint auf den ersten Blick unsinnig, weil der Assembler beim zweiten Marko-Aufruf eine Fehlermeldung gäbe. Wenn man "myGlobalLabel" aber mittels Makro-Parameter dynamisch konstruiert, kann man das Konstrukt zum Beispiel dazu verwenden, Kontroll-Strukturen mit Makros nachzubauen, zum Beispiel so etwas wie das Switch-Statement in C.

Weil es - bis auf Makros - nur einen, globalen Namensraum gibt, verwende ich weitere Konventionen, um Kollisionen zu vermeiden. Alle meine Namen für Register beginnen mit einem großen "R", alle meine Makro-Namen beginnen mit groß "M", bis auf Makros, die absichtlich wie Prozessor-Instruktionen aussehen sollen; siehe das Beispiel "ldiw" weiter unten. Die Großschreibung dient nur dazu, die Code-Lesbarkeit zu erhöhen, denn AVRASM2 unterscheidet groß/klein in der Standard-Einstellung nicht.

Labels für die Steuerung des Programmablaufs

Anders als in Hochsprachen, braucht man in Assembler immer viele Labels für Schleifen und Sprungziele. Weil Labels in Makros lokal sind, braucht man sich dort keine Mühe geben, eindeutige Namen zu erfinden.

Ich verwende deshalb in Makros immer die gleichen Schleifen-Labels. Meine Schleifen-Labels erläutern *nicht*, was die Schleife macht; das tun stattdessen Kommentare. Die Schleifen-Labels markieren vielmehr Anfang und Ende einer Schleife

Die Label können kurz sein, weil sie eh' nur pro Makro eindeutig sein müssen. Ich verwende "t" (top) für den Schleifenanfang, "b" (bottom) für das Schleifenende und "c" für ein "Continue"-Sprungziel. Ich hänge arabische Ziffern dran, falls in einem Makro mehrere Schleifen vorkommen. Beispiel:

```
...
;; algorithm to make sure I store neither the checksum nor the F7 byte
ld      r23,      z+      ; get first (previous) byte
rjmp    b1

t1:     st        x+, r23   ; store previous (first) byte
        inc      r25      ; byte counter
        breq     error    ; maximum 256 bytes
        mov     r23, r22   ; the next (second) becomes the previous (first)
b1:     ld        r22, z+   ; get second (next) byte
        cpi     r22, 0xF7 ; end?
        brne    t1

...
rjmp    exit

error:
...

exit:
```


Üblicherweise habe ich auch `exit` and `error` -Labels in einem Makro. `exit` steht immer am Makro-ende.

Obiges Beispiel zeigt auch die übliche Praxis, zunächst zum Schleifenende zu springen, wenn man zuerst eine Schleifenbedingung testen muss. Das ist effizienter, als die Schleifenbedingung am Anfang der Schleife zu testen; denn dann müsste am Ende der Schleife ein unbedingter Sprung sein, der mehr Zeit kostet, als der einmalige unbedingte Sprung zum Schleifenende.

7. Funktionen und Interrupt-Handler bauen

Wie man Makros nutzt, um Geschwindigkeit gegen Platz einzutauschen

Eine wichtige Eigenschaft von Makros ist, dass sie "inline" Code liefern. Der Code wird an der Stelle eingesetzt, an der im Assembler-Source-Code der Makro-Aufruf steht. Der Aufruf (invocation) bewirkt, dass der Assembler den Code-Block einfügt, der bei der Deklaration des Makros festgelegt wurde. Das geschieht also zur Entwicklungszeit und nicht zu Laufzeit, wie der Begriff "Aufruf" vielleicht nahe-läge. Ein Makro-Aufruf hat gegenüber einem Funktionsaufruf den Vorteil, dass der Prozessor keine call- und return-Instruktionen ausführen braucht, die wegen der Stack-Nutzung relativ langsam sind. Von Nachteil ist natürlich, dass die Codegröße mit jedem Makro-Aufruf wächst, während die Code-größe durch Funktionsaufrufe nur minimal wächst.

In meinen Projekten implementiere ich fast alle Funktionen mit Hilfe von Makros. Dadurch kann ich von Fall zu Fall entscheiden, ob eine Funktion besser ist oder "Inlining" durch das Makro.

Dazu lege ich den kompletten Funktionsrumpf in ein Makro ohne die `ret`-Instruktion. Wenn ich eine Funktion brauche, schreibe ich ein Label für die Funktions-Einsprung-Adresse, rufe das Makro auf und schreibe ein `ret` dahinter. Beispiel:

```
.macro      MtestFunction
.ldr       r24, 123

t:
...
cp        r24, r25
breq     t
...

exit:
.endm     ; MtestFunction

...
rcall    testFunction      ; function call
...

testFunction:                ; function definition
MtestFunction                ; with body delivered by macro invocation
ret
```

Dieses Vorgehen hat keinerlei Nachteile und hat den Vorteil, dass im Funktionsrumpf lokale Labels verwendet werden können, wie oben beschrieben.

Wenn Funktionen so konstruiert werden, hat man immer noch die Möglichkeit das Makro an bestimmten Stellen direkt aufzurufen, falls der Geschwindigkeits-Vorteil der Inline-Variante wichtig und der längere Code hinnehmbar ist:

```
...
MtestFunction                ; macro invocation = inlining
...
```

Ich setze die Funktions-Definitionen an das Ende des Source-Codes oder in eine `.asm-Include-Datei`, wenn die Funktion(en) Teil eines Moduls sind, das wiederverwendet werden kann. Als Konvention gebe ich dem Makro den gleichen Namen wie der Funktion, mit einem zusätzlichen "M" am Anfang. Mit dem "M" mache ich grundsätzlich Makros kenntlich und vermeide so auch Kollisionen mit gültigen

Assembler-Instruktionen, es sei denn, das Makro soll sich so "anfühlen" wie eine solche; siehe Makro "Idiw" weiter unten.

Sollte sich herausstellen, dass eine Funktion lexikalisch nur einmal aufgerufen wird, so lasse ich die Funktionsdefinition weg und setze einen Makro-Aufruf anstelle des Funktionsaufrufs.

Als Konsequenz daraus folge ich bei solchen Makros allen Aufruf-Konventionen und Register-Verwendungs-Konventionen wie bei Funktionen (siehe unten).

Makros in Include-Dateien

Ich kombiniere Makros mit dem System, den Code mit Hilfe von Include-Dateien zu strukturieren. Makros, die Funktionsrümpfe enthalten, gehen in .inc-Dateien. Funktionsdefinitionen gehen in .asm-Dateien.

Genauso gehe ich bei Interrupt-Handlern vor, mit den folgenden Unterschieden:

- Die Interrupt-Handler-Definitionen enden mit `rte` statt `rts`
- Makros, die den Rumpf des Interrupt-Handlers bilden, stehen in der .asm-Datei - dieselbe in der auch die Definition des Interrupt-Handlers steht - statt in einer .inc-Datei, denn Interrupt-Handler werden immer aufgerufen und können nicht "inline" sein.

Die Sichtbarkeit von Funktionen und Daten einschränken

Funktionen, die nur lokal in einem Makro bekannt sind

Funktionen, die nur lokal in einem Makro bekannt sind, können wie folgt konstruiert werden:

```
.macro      Mtest
ldi        R24, 123
...
rcall     myPrivateFunc1
...
rcall     myPrivateFunc2
...
rjmp     exit

myPrivateFunc1:
clr      r24
...
ret

myPrivateFunc2:
ldi      r24, 789
...
ret

exit:

        .endm      ; Mtest

Mtest
```

Die Label `myPrivateFunc1` and `myPrivateFunc2` sind nur im separaten Namensraum des Makros `Mtest` definiert. Deswegen können andere Funktionen mit demselben Namen anderswo existieren, ohne dass es zu einer Namenskollision kommt.

Achtung: auch in verschachtelten Makros, die innerhalb von solchen Makros wie zum Beispiel `Mtest` aufgerufen werden, sind solche Labels wie `myPrivateFunc1` nicht bekannt.

Variablen, die nur lokal in einem Makro bekannt sind

Wenn man das Konzept der lokalen Funktionen auf lokale Variablen (die man in C "static" nennt) ausdehnen will, stößt man auf eine Schwierigkeit:

```
.macro      MtestNew

.dseg
myPrivateVar:
.byte      1

.cseg
lds        R24, myPrivateVar

.endm      ; MtestNew
```

MtestNew

AVRASM2 gibt eine Fehlermeldung: "error: .byte directive illegal in macro definition". Variablen im RAM die lokal und privat in einem Makro sind, scheinen nicht möglich zu sein. Mit folgendem Konstrukt geht es aber doch:

```
.macro      MtestNew

.dseg
myPrivateVar:
.org        myPrivateVar +1

.cseg
sts        myPrivateVar, r24

.endm      ; MtestNew
```

MtestNew

Das ist vollkommen in Ordnung. Letztlich macht `.byte` nämlich auch nichts anderes, als den internen `.org`-Zeiger um die angegebene Anzahl Bytes weiterzusetzen.

Diese Konstruktion hat einen kleinen Nachteil: AVRASM2 führt Buch, wieviel Speicher das Projekt insgesamt braucht und gibt das am Ende in einer kleinen Tabelle aus. Die "Begin"-, "End"- und "Code"-Spalten sind korrekt, die "Data"-Spalte in diesem Fall nicht. Der Assembler scheint dazu nur aufzuzaddieren, wie viel `.byte` angegeben wurden und berücksichtigt `.org` nicht. Das heißt: das Programm ist richtig, die Data-Spalte in der Tabelle aber falsch. Man muss den Speicherverbrauch daher selbst aus Begin und End errechnen.

Auch lokale Variablen im EEPROM-Segment machen etwas Mühe, denn

```
.macro      Mtest
...
.eseg
myPrivateVar: .db      123

.endm      ; Mtest
```

gibt eine kryptische Fehlermeldung:

```
AVRASM: AVR macro assembler 2.2.6 (build 63 Apr 26 2016 14:42:08)
Copyright (C) 1995-2016 ATMEL Corporation
===> FATAL ERROR: Internal: malformed directive: myVar:      .db
Assembly failed, 1 errors, 0 warnings
```

Wenn man Label und `.db` auf getrennte Zeilen setzt, ist Alles gut:

```
.macro      Mtest
...
.eseg
myPrivateVar:
.db          123
```

```
.endm ; Mtest
```

8. Wozu kann man den C-artigen Pre-Processor nutzen?

AVRASM2 bietet zusätzlich C-artige Pre-Processor-Anweisungen. Allerdings scheint das kein eigener Verarbeitungsschritt vor den beiden Assembler-Passes zu sein, sondern diese Anweisungen werden vorab zusammen mit dem ersten Assembler-Pass verarbeitet.

Einige C-Pre-Processor-Anweisungen scheinen redundant mit ihren Assembler-Verwandten zu sein. Zum Beispiel dokumentiert Microchip, dass `#include` funktional gleichwertig ist mit `.include`.

`#define` scheint redundant mit dem Assembler `.equ` oder `.set`-Anweisungen. Die Unterschiede sind:

- Wenn man einen Wert per Assembler-Kommandozeile setzen will, kann man dort ein Pre-Processor-Makro setzen. Ein Assembler-Symbol kann so nicht gesetzt werden. Es ist verwirrend, aber im C-Pre-Processor nennen sich auch die einfachen `#defines`, die für Zahlen oder Strings stehen und keine Argumente auswerten, "Makros", obwohl sie sich eher wie Symbole "anfühlen".

Was Operationen, Funktionen und Ausdrücke anbelangt:

- Der Assembler kann mit Assembler-Symbolen und Zahlen rechnen. Das ist allerdings auf einige logische, boolesche und arithmetische Funktionen beschränkt und wird mit 64-Bit-Genauigkeit ausgeführt.
- AVRASM2-Assembler-Symbole können *nicht* für Strings stehen und es gibt auch keine Stringfunktionen, außer `strlen()`. Das Argument von `strlen()` muss entweder ein literaler String sein, also zum Beispiel "abc" - was nicht sehr nützlich scheint - oder ein Assembler Makro-Parameter - was schon nützlicher erscheint, oder ein C-Pre-Processor-Makro, was u.U. auch nützlich ist.
- Der C-Pre-Processor kann *nicht* rechnen, auch wenn das oft so aussieht (das macht dann der Assembler oder der Compiler bei C). Nur die Bedingung bei `#if` rechnet er doch aus. Auf der anderen Seite hat er einige wenige String-Funktionen
- Die C-Pre-Processor-String-Funktionen beschränken sich auf Zusammenfügen (concatenation), Umwandeln in ein String-Literal (stringification) und Muster-Erkennung (pattern matching). Man kann mit diesem begrenzten Vorrat doch viele erstaunliche Dinge tun. Unglücklicherweise arbeitet der C-artige Pre-Processor in AVRASM2 nicht so, wie man es von einem Standard-C-Pre-Processor erwartet. In Anhang 1 steht Beispiel-Code, der das zeigt.

Diese Beschränkung von AVRASM2 in den String-Funktionen und der Mustererkennung hat zur Folge, dass einige fortgeschrittene Dinge nicht funktionieren, wie zum Beispiel X-Macros https://en.wikipedia.org/wiki/X_Macro oder der Nachbau von Arithmetik-Funktionen.

Pre-Processor-Makros überlappen funktional teilweise mit Assembler-Makros (`.macro`). Zum Beispiel:

- `.macro`-Argumente können numerisch sein, Symbole oder Literale oder String-Literale (mit " als Begrenzer)
- Pre-Processor-Makro-Argumente können ebenfalls numerisch sein, Symbole oder Literale oder String-Literale (mit " als Begrenzer)
- Assembler-Makro-Argumente können miteinander oder mit Literalen zusammengefügt werden. Diese Literale können allerdings keine Ziffern sein (der Grund dafür ist unklar)
- Pre-Processor-Makro-Argumente können ebenfalls zusammengefügt werden. Dafür gibt es den Concatenation-Operator `##`. Die Regeln dafür sind unübersichtlich, aber gut dokumentiert. Leider weicht AVRASM2 dort ab, wie oben erwähnt.

Letztlich ist es oft Geschmacksache, welche der beiden Makro-Konstrukte man verwendet. Ich versuche, soweit wie möglich mit Assembler-Anweisungen zu kommen und verwende den C-Pre-Processor nur in Ausnahmefällen, zum Beispiel um sicherzustellen, dass derselbe Code nur einmal eingebunden wird. Zum Beispiel in der Datei `trUARTS.inc`:

```
;;; file trUARTS.inc
;;;
#ifdef _TRUARTS_INC_
```

```

#define _TRUARTS_INC_
...

#endif /* _TRUARTS_INC_ */

;; ***** END OF FILE *****

```

Diese Technik wird meist in C-Include-Dateien eingesetzt (zum Beispiel in denen der C-Library). Der gleiche Effekt könnte auch mit Assembler-Anweisungen erreicht werden. Dabei würden dann aber Symbole wie `_TRUARTS_INC_` in der Assembler-Symbol-Tabelle und in der Map-Datei auftauchen, was ich nicht möchte. Deshalb ziehe ich hier die C-Pre-Processor-Variante vor.

9. Lexikalische Details

Ich versuche, in der Schreibweise soweit als möglich kompatibel zu `avr-as` zu bleiben. Deswegen schreibe ich alle Assembler-Anweisungen klein, obwohl sie in der Microchip-Dokumentation groß stehen. `AVRASM2` ist in der Standard-Einstellung groß/klein egal.

Wegen `avr-as`-Kompatibilität schreibe ich auch `.endm` statt `.endmacro`.

Kommentare und Einrückungen schreibe ich so, wie es der AS-Mode von EMACS nahelegt.

10. Prozessor-Ressourcen optimal nutzen

Wann den Stack nutzen?

`push-` und `pop-`Instruktionen von AVR-Prozessoren sind vergleichsweise effizient; sie brauchen nur zwei Taktzyklen. Das ist wichtig, weil man Prozessor-Register oft auf dem Stack sichern muss. Das ist Pflicht bei Interrupt-Handlern und kann auch in normalen Funktionen erforderlich sein, wenn man Register sichern muss, für aufrufende oder aufgerufene Funktionen (siehe Register-Nutzungs-Konvention unten).

Falls man nicht genügend Register zu Verfügung hat, ist das übliche Vergehen, kurzlebige Variablen auf dem Stack anzulegen (siehe sogenannte "auto"-Variablen in C). Dazu braucht man einen sogenannten Frame-Zeiger (frame pointer), der auf den allozierten Stack-Bereich zeigt. Eine Alternative dazu gibt es auf AVR-Prozessoren nicht, da der Stack-Pointer nicht als Register zu Verfügung steht, geschweige denn für register-relative Adressierung (laut Microchip "data indirect with displacement addressing" genannt). Zwei Register (genauer: Registerpaare) können das: Y und Z. `avr-as` nutzt dafür Y, was plausibel ist, weil Z noch für andere Dinge gebraucht wird, die Y nicht unterstützt.

Register-Inhalte mittels Y auf dem Stack sichern oder von dort laden braucht auch zwei Takte. Das "Displacement" (Offset) kann zwischen 0 und 63 Bytes liegen und ist immer positiv.

Allerdings braucht der Prozessor relativ viel Zeit, Platz auf dem Stack anzulegen oder wieder freizugeben. Denn: der Stack ist nicht als "normales" Registerpaar implementiert, sondern als I/O-Register. I/O-Register können aber nur 8-Bit-weise manipuliert werden. Damit man den Stack 16-Bit-weise sicher manipulieren kann, muss man die Interrupts kurzzeitig ausschalten. Das kostet extra Zeit und Platz im Programmspeicher.

Daher versuche ich immer, soweit als möglich nur mit Prozessor-Registern auszukommen und vermeide Variablen auf dem Stack.

Wenn ich Arrays brauche, prüfe ich, ob sie wirklich dynamisch angelegt und wieder aufgegeben werden müssen oder ob ich nicht doch genügend RAM (`.dseg`) zur Verfügung habe, um sie statisch anzulegen.

Wegen Interrupts und Interrupt-Handlern: Ich sichere Register immer auf dem Stack. Es gibt auf AVR-Prozessoren noch die Möglichkeit, Register in anderen Registern zu sichern, die extra dafür freigehalten werden. Das bietet einen kleinen Geschwindigkeits-Vorteil gegen über `push-` und `pop-`Instruktionen. Diese Option funktioniert allerdings nur, weil auf AVR-Prozessoren Interrupts nicht von anderen Interrupts unterbrochen werden, es sei denn man schaltet diese Möglichkeit selbst in Interrupt-Handlern an.

Auf Strukturen und Arrays zugreifen

Abgesehen von allereinfachsten Fällen, muss man Structs und Arrays im RAM anlegen.

Beispiel für eine Struktur in C-Syntax:

```
struct point {
    int x;
    int y;
};
```

die man zum Beispiel so nutzt:

```
struct point pt;
pt.x = 456
pt.y = 789
```

Falls der Integer-Datentyp zwei Byte braucht, könnte man in Assembler so schreiben:

```
;;define structure
.dseg
ptxlo: .byte 1
ptxhi: .byte 1
ptylo: .byte 1
ptyhi: .byte 1

.cseg
ldi r24, low(456)
ldi r25, high(456)
sts ptxlo, r24
sts ptxh, r25
ldi r24, low(789)
ldi r25, high(789)
sts ptylo, r24
sts ptyh, r25
```

Die Adressierung mit absoluten Adressen - wie hier im Beispiel mit `sts` wird von Microchip "Data Direct" genannt. Der Code ist korrekt, aber die Tatsache, dass es sich um eine Struktur handelt, ist nur durch die Variablennamen erratbar.

Folgende Programmierung finde ich klarer (die verwendeten Makros findet man in Anhang 3):

```
;; declare structure offsets
MtrSetOffset 0
MtrNewOffsetSymbol POINT_X_OFFSETS, 2
MtrNewOffsetSymbol POINT_Y_OFFSETS, 2
;; length of a structure instance
.equ POINT_LEN = TR_OFFSETS

;; allocate RAM = define structure
.dseg
pt: .byte POINT_LEN

.cseg
ldi r24, low(456)
ldi r25, high(456)
sts pt+POINT_X_OFFSETS+0, r24
sts pt+POINT_X_OFFSETS+1, r25
ldi r24, low(789)
ldi r25, high(789)
sts pt+POINT_Y_OFFSETS+0, r24
sts pt+POINT_Y_OFFSETS+1, r25
```

Diese Schreibweise ist vor allem dann klarer, wenn man mehrere Instanzen der gleichen Struktur anlegt, einzeln oder als Array. Mit der naiven Schreibweise wird pro Instanz ein weiterer Satz Symbole fällig, während bei der Programmierung mit Offsets pro Instanz nur ein weiteres Symbol hinzukommt.

Wohlgemerkt, der erzeugte Maschinen-Code ist identisch! Es geht darum, wie verständlich, wartbar und leicht erweiterbar der Code ist.

Muss indiziert zugegriffen werden - wie bei Arrays - ist die Schreibweise mit Offsets unabdingbar und ihr Vorteil wird deutlich, weil statische und dynamische Strukturen syntaktisch ähnlich abgebildet werden:

```

ldiw    z, pt                ; load pointer register with address
ldi     r24, low(456)
ldi     r25, high(456)
std     z+POINT_X_OFFS+0, r24
std     z+POINT_X_OFFS+1, r25
ldi     r24, low(789)
ldi     r25, high(789)
std     z+POINT_Y_OFFS+0, r24
std     z+POINT_Y_OFFS+1, r25

```

Der Zugriff über ein Zeigerregister mit Offset - wie hier im Beispiel per `std` unter Verwendung von Register `z` - nennt Microchip "Data Indirect with Displacement".

Wäre `pt` ein Array, zum Beispiel mit `PT_ARR_NUM` Elementen

```

.dseg
ptArr: .byte    PT_ARR_NUM*POINT_LEN

```

so könnte nach

```

    adiw    z, POINT_LEN

```

auf das nächste Array-Element zugegriffen werden.

Bei AVR-Prozessoren ist der Zugriff auf das RAM übrigens nur wenig langsamer als der Zugriff auf Register. Absolute Adressierung per `lds` bzw. `sts` braucht zwei Clock-Zyklen. Die indizierten Zugriffe per `ld` bzw. `st` sowie indiziert mit Offset per `ldd` bzw. `std` sowie mit pre-decrement oder post-increment brauchen ebenfalls zwei Clock-Zyklen. Ausnahme: `st x, Rr` braucht nur einen statt zwei.

Das bedeutet, dass man bei AVR-Prozessoren fast keinen Geschwindigkeits-Vorteil durch den Einsatz von Zeiger-Registern hat und man deswegen bei statischen Strukturen ruhig mit absoluter Adressierung arbeiten kann, wie oben gezeigt. Allerdings braucht absolute Adressierung wegen der Adresse immer ein Programmspeicherwort mehr als indizierter Zugriff.

Structs als Teil statischer Objekte

Wenn man mehrere Instanzen von Structs anlegen will, muss man für jede Instanz RAM-Speicher reservieren und ein Symbol erzeugen, das für die Speicher-Anfangsadresse steht. Statt dies jedes Mal einzeln mit `.dseg` hinzuschreiben, kann man das zum Beispiel mit folgendem Konstrukt vereinfachen:

```

.macro    MtrFIFONew
...

.dseg                                ; switch to data section
dummy:                                     ; dummy local symbol in dseg
.equ     DSEGtrFIFO@0 = dummy ; proper global symbol
.org     DSEGtrFIFO@0 + TRFIFO_BUF_OFFS + @1 ; allocate space

.cseg                                ; switch back to code
...                                     ; initialise instance

.endm    ; MtrFIFONew

```

Das komplette Makro ist in Anhang 4 gelistet. Wie oben beschrieben, erzeugt der Label "dummy" ein Symbol, das nur lokal im Makro bekannt ist. Der Name "dummy" sagt schon, dass ich dieses Symbol nicht weiter verwende. Mit dem folgenden `.equ` erzeuge ich ein globales Symbol mit dem gleichen Wert, über das die Methoden des FIFO-Objects später auf diesen Speicherbereich zugreifen werden. Dazu sorgt der Makro-Parameter `@0` für ein eindeutiges Symbol für diese Instanz. Beim Makro-Aufruf gebe ich so als erstes Argument ein Literal an, mit dem ich die Instanz fortan identifiziere:

```

MtrFIFONew UARTrx, 256

```

Das zweite Makro-Argument gibt in diesem Beispiel die Pufferlänge an.

Die Zugriffsmethoden auf das FIFO sind ebenfalls als Makros implementiert, zum Beispiel

```

MtrFIFOgetNxtByte    UARTrx

```

Ich plane, den ganzen FIFO-Code unter GPL zu veröffentlichen.

Sonderfall EEPROM

Das EEPROM kann auf AVR-Mikrocontrollern nur über spezielle I/O-Register zugegriffen werden. Der Zugriff dauert einige Taktzyklen und überdies muss man noch warten, falls noch ein Schreibzugriff hängig sein sollte.

Daher kann es sinnvoll sein, Daten vom EEPROM ins RAM zu kopieren.

Welche Register für was verwenden?

Die üblichen AVR-Mikrocontroller, wie zum Beispiel ATmega, haben 32 Register. Der Assembler-Programmierer hat volle Freiheit sie zu nutzen, für was immer er will.

Ich beschloss, das Rad nicht neu zu erfinden, sondern die Register-Nutzungs-Konvention des AVR-GNU-C-Compilers (GCC) zu übernehmen (siehe Aufstellung in Anhang 2):

Als Faustregel nutze ich die Register r18 - r27 und r30, r31 für kurzlebige Dinge (in C wären das "auto"-Variablen), weil einige der Register eh' zur Funktions-Argument-Übergabe vorgesehen sind. Deshalb nutze ich r1 - r17 für längerlebige Dinge (in C wären das "static"-Variablen). Um nachzuhalten, welche Register ich für was verwende, benutze ich meist symbolische Registernamen, die mittels `.def` definiert und mittels `.undef` wieder freigegeben werden. So würde mich der Assembler warnen, wenn ich ein Register gleichzeitig für zwei Zwecke verwendete.

Ich verwende Registernamen aber auch für kurzlebige Dinge, damit der Code lesbarer wird. Die `.def` und `.undef` -Anweisungen kommen dann in Paaren, zum Beispiel:

```
.macro      MtrUARTrxInterrupt
.def        scratch      = r25
.def        data         = r24
push        scratch
LOAD        scratch, SREG
push        scratch      ; save status reg
push        data
...
LOAD        data, UDR@0  ; get data as fast as possible to make
                        ; room in the hardware buffer
...
pop         data
pop         scratch      ; pop status from stack
STORE      SREG, scratch ; restore status register
pop         scratch
.undef     data
.undef     scratch
.endm      ; MtrUARTrxInterrupt
```

Die Register r26 bis r31 - auch x, y, z genannt - unterstützen spezielle Adressierungsarten. Dabei gibt es zwischen x, y, und z Unterschiede, welche Adressierungsarten unterstützt werden:

Data Indirect with Displacement (LDD, pointer access with 0-63 offset):	Y, Z
Data Indirect with Pre-Decrement or Post-Increment(LD):	X, Y, Z
Program Memory Constant Addressing (LPM, ELPM, SPM):	Z
Program Memory with Post-Increment (LPM Z+, ELPM Z+):	Z
Indirect Program Addressing (IJMP, ICALL):	Z

Neben x, y, z unterstützt auch das Registerpaar r25:r24 16-Bit Operationen wie z.B. ADIW .

11. Program-Ablauf und Gleichzeitigkeit

Hardware arbeitet oft parallel

Moderne Mikrocontroller bieten eine Fülle von eingebauter Hardware, die unabhängig vom und parallel zum Prozessor laufen kann, zum Beispiel ADC, Timer, UART, TWI, SPI. Manche Mikrocontroller

haben sogar mehrere solcher Einheiten, zum Beispiel zwei oder gar vier UARTs. Die Hardware-Einheiten bieten üblicherweise einen Polling-Modus (Abfrage-Modus) und einen Interrupt-Modus, um mit dem Prozessor zu kommunizieren. Der Interrupt-Modus ist meist effizienter, weil keine Prozessor-Zeit mit Warten verschwendet wird. Allerdings ist Interrupt-Betrieb komplexer zu planen und umzusetzen.

In anderen Umgebungen kümmert sich das Betriebssystem um "low-level" Hardware-Funktionen und Interrupts und bietet für die Applikation geeignete Schnittstellen, die uns von den Details abschirmen. Auf kleinen Mikrocontrollern ist oft kein Platz für ein Betriebssystem neben der Applikation. Neuere Controller-Modelle bieten so viel Programmspeicher, dass ein Betriebssystem in den Bereich des Möglichen rückt. Für AVR-Controller, wie zum Beispiel ATmega, scheint es aber bislang keinen bekannten de-facto-Standard für ein Betriebssystem zu geben.

Wie mehrere Tasks quasi gleichzeitig ablaufen können

Sogar einfache Mikrocontroller-Anwendungen müssen mehrere Dinge tun, die dem Benutzer gleichzeitig erscheinen. Zum Beispiel eine Tastatur oder andere Eingabe-Geräte abfragen, auf ein Display schreiben, etc. Auch das wird uns sonst von einem Betriebssystem abgenommen.

Wenn man kein Betriebssystem hat, ist die einfachste Strategie, die einzelnen Aufgaben (Tasks) nacheinander reihum ablaufen zu lassen. Wenn eine Task nichts zu tun hat, lässt sie einfach die nächste ran.

Programmfluss-Beispiel

Die Struktur, die ich nutze, folgt folgenden Prinzipien:

- Interrupt-Handler sollen so kurz wie möglich sein
- Interrupts kommunizieren mit dem Hauptprogramm nach einem Produzent-Verbraucher-Paradigma (producer consumer paradigm). Oft ist ein FIFO ein geeignetes Mittel dafür
- Das Hauptprogramm besteht aus "Tasks"
- Die Tasks werden reihum abgearbeitet
- Jede Task prüft zunächst, ob es für sie was zu tun gibt. Falls nicht, gibt sie die Kontrolle unverzüglich an das Hauptprogramm zurück und das gibt die Kontrolle an die nächste Task
- Diese Prüfungen müssen so schnell und effizient wie möglich sein, weil sie direkt von der zur Verfügung stehenden Gesamtleistung abgehen

In der Regel habe ich dann die folgende Programm-Struktur:

- Reset and Interrupt vectors
- Interrupt handler(s)
 - M...ISR1
 - M...ISR2
 - ...
 - M...ISRN

Reset:

- Initialisation
- Hauptprogramm

mainloop:

- M...Task1
- M...Task2
- ...
- M...TaskM
- rjmp mainloop

Sowohl Interrupt-Handler also auch Task sind als Makros geschrieben. Interrupt-Handler bestehen noch aus einem Label und abschließendem `rte` nach dem Makro-Aufruf. Beispiel:

```
trUARTTxInterrupt0:
```

```

        MtrUARTTxInterrupt 0
        rte

trUARTTxInterrupt1:
        MtrUARTTxInterrupt 1
        rte

```

Das Beispiel zeigt die Empfangs-Interrupt-Handler für einen ATmega mit *zwei* UARTS. Der Code für die zwei UARTs ist also doppelt vorhanden und entsteht aus demselben Makro, wobei ein Makro-Argument die physische UART-Instanz angibt.

Tasks sind inline und keine Funktionen, weil sie lexikalisch eh' nur einmal erscheinen und Funktionen nur zusätzlichen "Overhead" brächten. Beispiel:

```

mainloop:
        MhandleTWIrxFixTask

        MhandleMIDIpRxTask

        MhandleTWITxTask

        MhandleKeyPadRxTask

        rjmp      mainloop

```

Die Task-Makros prüfen gleich am Anfang auf anstehende Arbeit und springen sofort zum Makro-Ende, falls es nichts zu tun gibt.

Dieses Schema ist einfach und reicht aus, solange dieses Reihum-Abarbeiten fair genug ist, für die jeweilige Anwendung.

12. Tipps und Tricks

Wie man Code kürzer und lesbarer machen kann

Beispiel-Makro:

```

        .macro      ldiw
        ;;;;;;;;;;;;;;
        ;; Macro to simplify loading a RAM address into x, y, or z
        ;; marco arg1=@0:  either x, y, z register to load with...
        ;; marco arg2=@1:  ...address
        ;;
        ldi        @0L,  LOW(@1)
        ldi        @0H,  HIGH(@1)
        .endm      ; ldiw

```

Anwendung:

```

        ldiw      z,  bufferAddress

```

Das ist ein typisches Makro, das eine AVR-Instruktion sein *könnte*. Sollte Microchip eines Tages eine solche Instruktion zum Prozessor-Instruktionssatz hinzufügen, würde man einfach die Makro-Deklaration streichen.

Portierbarkeit

Es ist immer eine gute Idee Makros zu übernehmen, die andere Leute geschrieben und getestet haben.

Ein Beispiel ist macros.inc von Atmel. Die Makros in macros.inc sind beschrieben in der Atmel Application Note AVR001: Conditional Assembly and portability macros - doc2550.pdf. Die Portability-

Makros abstrahieren I/O-Prozessor-Instruktionen und helfen so Code zu schreiben, der auf vielen AVR-Controllern ohne Änderung läuft.

Ich habe zu macros.inc nur die üblichen Pre-Processor-Anweisungen hinzugefügt

```
...
#ifdef _MACROS_INC_
#define _MACROS_INC_
...
#endif /* _MACROS_INC_ */
```

um zu vermeiden, dass die Include-Datei mehrfach verarbeitet wird.

13. Beispiel für eine Entwicklung-Umgebung unter Windows

Ich entwickle unter Windows 10 Pro 64-Bit und nutze AVRASM2 von der Kommandozeile oder aus EMACS heraus. Dazu habe ich das zugehörige Unterverzeichnis im passenden Ordner von "Program Files" "installiert" (bzw. Program Files (x86) bei 64-Bit) nebst einem Eintrag in der Windows-Registry.

Auf einem weiteren PC nutze ich AVRStudio 4.19 unter Windows 10 Pro 32-Bit mit AVRISP MKII, um die AVR-Controller zu programmieren.

Auf beiden Maschinen verwende ich AVRASM2 Version 2.2.6 (build 63 Apr 26 2016 14:42:08). Ich bearbeite den Source-Code mit GNU EMACS Version 25.1 mit CUA-Keys und AS-Mode, den ich etwas an meine Vorlieben angepasst habe.

EMACS habe ich auf OneDrive "installiert". Dadurch finde ich auf beiden PCs immer die gleiche Umgebung vor.

Mit dem EMACS-serial terminal-Modus lasse ich mir gegebenenfalls Debug-Ausgaben anzeigen, die von einer AVR-UART-Schnittstelle kommen.

14. Referenzen

AVRASM2 Manual:

<http://ww1.microchip.com/downloads/en/DeviceDoc/40001917A.pdf>

AVR Instruction Set Manual:

<http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-0856-AVR-Instruction-Set-Manual.pdf>

AVR-Tutorials auf mikrocontroller.net mir Links zu Assembler-Dokumentation:

<https://www.mikrocontroller.net/articles/AVR#Tutorials>

15. Anhang 1 - Unzulänglichkeiten des C-Pre-Processors

Der folgende Beispiel-Code zeigt, dass der C-artige Pre-Processor, der in AVRASM2 eingebaut ist, nicht die gleiche Funktionalität hat, wie der C-Pre-Processor von avr-as (GNU AS), im Hinblick auf String-Funktionen, Pattern-Matching und Symbol-Ersetzung:

```
/* *****
/* This code has been run through AVR GCC {gcc version 4.8.1
(AVR_8_bit_GNU_Toolchain_3.4.5_1522)}
and AVRASM2 {AVR macro assembler 2.1.17 (build 435 Apr 10 2008 09:27:55)}
with the command lines
> avr-gcc -E testpp.c
The -E options runs pre-processing only and put output to stdout.
And
> avrasm2 -m map.txt testpp.c
The map file map.txt shows the names of the two generated assembler symbols and
their value in hex
*/

/* Example 1
Works as expected: GCC preprocessor and AVRASM2 preprocessing give the same result
*/

#define CRSYMBOL(sym, val) .equ sym = val
```

```

#define VALUE 123

        CRSYMBOL(symname, VALUE)

/* this expands to
   .equ symname = 123
as expected */

/* Example 2
   Works as expected in GCC preprocessor. However AVRASM2 does not expand VARPOSTFIX,
hence is /not/ compliant to C pre-processor standards.
*/

#define CAT_(a,b) a ## b
#define CAT(a,b) CAT_(a,b)
#define VARPOSTFIX 3

        .equ CAT(symname, VARPOSTFIX) = VALUE

/* The GCC preprocessor expands this to
   .equ symname3 = 123
as expected.
However AVRASM2 expands this to
   .equ symnameVARPOSTFIX = 123
which is /not/ what we expect */

/* the following nop is just there to keep AVRASM2 from complaining about an "empty
source file" */
        nop

/*****

```

16. Angang 2 - Konvention, wie Register verwendet werden

Die folgende Tabelle zeigt, welche Prozessor-Register für was vorgesehen sind. Ich habe das aus AVR GCC-Dokumentation übernommen:

```

;;;;;;;;;;;;;; Register Usage ;;;;;;;;;;;;;;
;;;
;;; I use any of registers r0 - r17 only via .def !
;;;
;;;Reg.   Function   Saving           Return Value
;;;-----
;;;R0     Temp Register   Scratch
;;;R1     Always 0     Callee must clear
;;;R2           Callee must save
;;;R3           Callee must save
;;;R4           Callee must save
;;;R5           Callee must save
;;;R6           Callee must save
;;;R7           Callee must save
;;;R8     Next Arg     Callee must save
;;;R9     Next Arg     Callee must save
;;;R10    Next Arg     Callee must save
;;;R11    Next Arg     Callee must save
;;;R12    Next Arg     Callee must save
;;;R13    Next Arg     Callee must save
;;;R14    Next Arg     Callee must save
;;;R15    Next Arg     Callee must save

;;;R16    Next Arg     Callee must save
;;;R17    Next Arg     Callee must save
;;;R18    Next Arg     Scratch
;;;R19    Next Arg     Scratch

```

```

;;;R20    Next Arg  Scratch
;;;R21    Next Arg  Scratch
;;;R22    Left most arg:      Scratch          Long byte0
;;;      Long byte0
;;;R23    Left most arg:      Scratch          Long byte1
;;;      Long byte1
;;;R24    Left most arg:      Scratch          Char
;;;      Char          Int low byte
;;;      Int low byte          Long byte2
;;;      Long byte2
;;;R25    Left most arg:      Scratch          Char: 0 or sign ext.
;;;      Int high byte          Int high byte
;;;      Long byte3          Long byte3
;;;R26    X:          Scratch
;;;R27    X:          Scratch
;;;R28    Y: Frame ptr.      Callee must save
;;;      Low Byte
;;;R29    Y: Frame ptr.      Callee must save
;;;      High Byte
;;;R30    Z:          Scratch
;;;R31    Z:          Scratch

```

17. Anhang 3 - Makros, um mit Strukturen umzugehen

AVRASM2 bietet von sich aus keine Anweisungen, um Structure-Offsets zu berechnen. Mit den folgenden beiden Makros kann man das aber leicht selbst nachrüsten:

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Macros to generate symbols for structure member offsets
;;;
;;; Usage:
;;;      MtrSetOffset is used to set the offset "counter",
;;;      specifically to reset it
;;;      MtrNewOffsetSymbol generates a new global symbol to be used as
;;;      a structure member offset
;;;

        .macro      MtrSetOffset
        .set        TR_OFFSET = @0
        .endm      ; MtrSetOffset

        .macro      MtrNewOffsetSymbol
        .equ        @0          = TR_OFFSET
        .set        TR_OFFSET = TR_OFFSET + @1
        .endm      ; MtrNewOffsetSymbol

```

18. Anhang 4 - Makro, um einen neue FIFO-Instanz anzulegen

Auszug aus der Datei trFIFO.inc. Das Makro erzeugt eine neue FIFO-Instanz im Speicher und initialisiert sie:

```

;;; Create data structure offsets. These are the same for all FIFO instances
MtrSetOffset      0
;;; structure offsets of the indices
MtrNewOffsetSymbol TRFIFO_HEAD_OFFS, 1
MtrNewOffsetSymbol TRFIFO_HDBT_OFFS, 1
MtrNewOffsetSymbol TRFIFO_TAIL_OFFS, 1
MtrNewOffsetSymbol TRFIFO_TLBT_OFFS, 1
;;; also BUF_OFFS is the same for all FIFOs
MtrNewOffsetSymbol TRFIFO_BUF_OFFS, 0

        .macro      MtrFIFOnew
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Macro to create a new FIFO instance

```

```

;;; - the /instance/ FIFO symbols, i.e. static data specific to the instance
;;; - reserve space in dseg for instance
;;; - initialise instance data structure.
;;; The FIFO buffer proper in dseg is also instantiated but is /not/ cleared
;;;
;;; Macro arg1:      Name of the FIFO instance
;;; Macro arg2:      FIFO size in bytes (must be 2, 4, 8, 16, 32, 64, 128 or 256)
;;; Macro arg3:      Packet size in bytes (must be a power of 2 and < FIFO size)

    ;; check the macro arguments
    .if      @1 > 256 || @1 < 2
      .error "FIFO size must be <= 256 and >= 2"
    .endif
    .if      exp2(log2(@1)) != @1
      .error "FIFO size must be a power of 2"
    .endif
    .if      @2 >= @1 || @2 < 1
      .error "Packet size must be smaller than FIFO size and >= 1"
    .endif
    .if      exp2(log2(@2)) != @2
      .error "Packet size must be a power of 2"
    .endif

    ;; static characteristics specific to this FIFO instance,
    ;; later accessed as immediate values
    .equ     TRFIFO_SIZE_@0      = @1
    .equ     TRFIFO_MASK_@0     = TRFIFO_SIZE_@0-1
    .equ     TRFIFO_PACKET_LEN_@0 = @2

dummy:
    .dseg                                ; switch to data section
                                ; dummy local symbol in dseg to capture the dot
    .equ     DSEGtrFIFO@0        = dummy    ; proper global symbol =
                                ; address of the instance
    .org     DSEGtrFIFO@0 + TRFIFO_BUF_OFFS + @1    ; allocate space for
                                ; instance in dseg

    .cseg                                ; switch back to code
    ;; set generic variables to zero = FIFO is empty.
    ;; The FIFO buffer is /not/ cleared
    ;; TRFIFO_BUF_OFFS is the same as the length of the generic part
    MtrClrMem DSEGtrFIFO@0, TRFIFO_BUF_OFFS

    .endm      ; MtrFIFONew

```

19. Anhang 5 - Beispiel-Projekt: Software für eine Synthesizer-Bedienoberfläche

Ich habe dieses Dokument geschrieben, nachdem ich meine eigene Bedienoberfläche (HUI) gebaut und programmiert hatte, die meinen do-it-yourself Analog-Synthesizer ansteuert. Das HUI hat folgende Eigenschaften:

- Hat Drehencoder und Drucktaster zu Eingabe und LEDs als Ausgabe
- Unterstützt MIDI
- Nutzt einige ATmega8 und ATmega644P

Siehe <http://web222.webclient5.de/prj/MusicEI/GTHL1/index.htm>

* * *