1 Entwurfsumgebung

1.1 Anforderungen

An ein µC-System werden fast immer zwei Anforderungen gestellt:

1. Möglichst geringer Speicherbedarf zur Kostenreduktion (Kosten pro Stück)
2. Echtzeitfähigkeit (garantierte Reaktionszeiten zur Laufzeit)

Falls die Anwendung auch noch aus Komplexitätsgründen einen 32 Bit µC erfordert, dann kann man davon ausgehen, dass noch eine dritte Anforderung dazukommt:

3. Gute Unterstützung der HW durch fertige Laufzeitbibliotheken (NRE-Kosten)

Die verfügbare Rechenleistung ist dagegen aktuell (2017) häufig kein ausschlaggebendes Kriterium mehr. Für die Echtzeitfähigkeit wird in einem µC-System meist auf das Konzept der ereignisgesteuerten Unterbrechungen (dazu später ein eigenes Kapitel) zurückgegriffen. Die damit erzielbare Reaktionszeit reicht bei geschickter Aufteilung des Gesamtprogramms selbst bei eher langsamen µC (Taktfrequenz unter 20 MHz) noch für sehr viele praktische Anwendungen aus.
Zudem kann in wirklich kritischen Fällen die Rechenleistung durch den Einsatz mehrerer Kerne oder spezieller HW (DSP-Einheiten, Fließkomma-Einheiten) mit moderaten Stückkosten erhöht werden.

Prinzipiell könnten C- Programme für einen µC genauso entwickelt werden wie für einen PC. Das setzt dann allerdings voraus, dass auch die Funktionen der Standardbibliothek sowie die einiger anderer Bibliotheken (z. B. für die dynamische Speicherverwaltung) zur Verfügung stehen. Falls sich auf dem µC auch ein Betriebssystem wie z.B. Linux befindet, ist das auch tatsächlich der Fall und dann unterscheidet sich die Programmentwicklung in der Tat nur gering von der für einen PC.
Dann ist aber auch ein vollwertiges Betriebssystem erforderlich und das alleine benötigt meist mehr Speicher (Programm- und Datenspeicher) als die eigentliche Anwendung. Ein aktuelles (2016) Embedded Linux benötigt ca. 1 Mbyte Programmspeicher und auch ca. 1 Mbyte Datenspeicher nur für sich selbst.

1.2 Softwarearchitektur

Programme für einen µC unterschieden sich selbst dann, wenn ein Betriebssystem vorhanden ist, noch in einigen Eigenschaften von Programmen auf einem PC. Das sind u.a.:

- Ereignisbearbeitung (mittels ISRs) am Betriebssystem vorbei
- Eingeschränkter Gebrauch von üblichen Bibliotheken (z.B. kein Dateisystem)
- Abhängigkeit von der HW (keine oder sehr geringe Portabilität)
- Kein oder geringer Schutz bei Programmfehlern

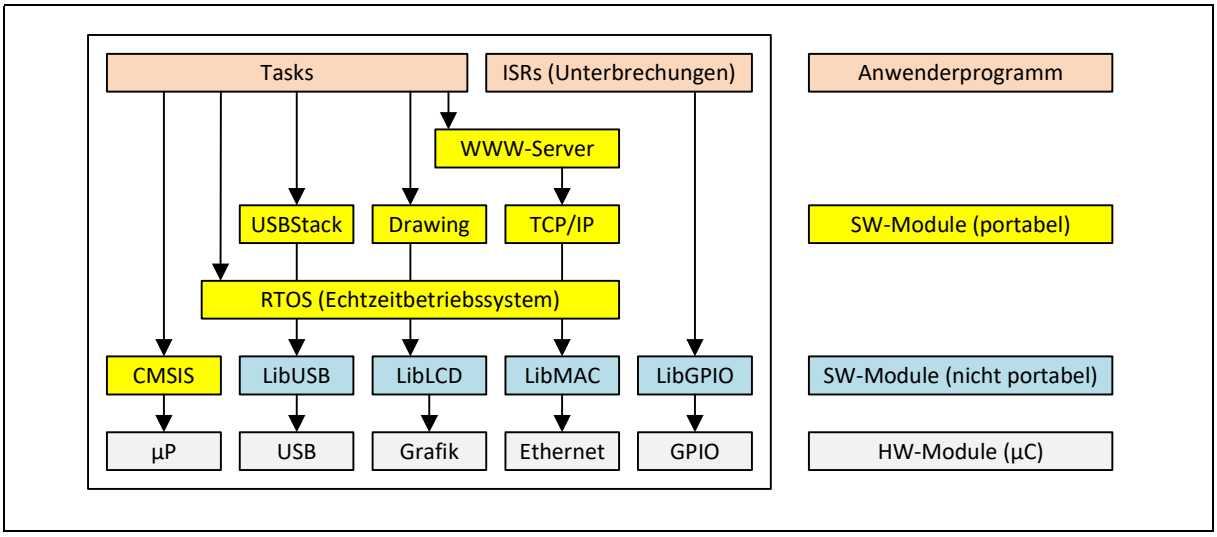

Abbildung 1: Softwarearchitektur µC

In Abbildung 1 ist eine recht typische Softwarearchitektur für eine Anwendung zu sehen, die auf einem µC mit einem spezialisierten Betriebssystem läuft.
Auf der untersten Ebene sind einige Hardwaremodule des µC eingezeichnet. Der Rechenkern selbst wird hier mit dazugezählt. Auf der obersten Ebene sind die Programmteile zu sehen, die vom Entwickler selbst geschrieben werden müssen. Hier ist auch die typische Trennung in einen ereignisgesteuerten Teil (ISRs) und einen sequentiell ablaufenden Teil (Tasks) zu sehen. Bei sehr kleinen Anwendungen genügen diese beiden Schichten schon. Der Anwender bedient dann direkt die HW-Module, indem er in die entsprechenden Register Werte schreibt oder von dort liest.
Diese Methode ist jedoch zum einen fehleranfällig (z.B. weil man eine wesentliche, aber nicht leicht zu findende Information in der Dokumentation übersieht) und das Programm wird auch nur auf genau diesem µC lauffähig sein.
Daher bieten inzwischen alle Hersteller Bibliotheken in C an, mit denen der Anwender etwas abstrakter mit der zugrundeliegenden Hardware arbeiten kann. Dies ist in Abbildung 1 die zweite Schicht von unten. Pro HW-Modul findet man typischerweise genau eine Sammlung von Funktionen, die dann die abstrahierten Aufrufe (setze GPIO x auf Wert y) in die passenden Registeroperationen (schreibe Wert y in das Register x und achte dabei darauf, dass sonst nichts verändert wird) umsetzt. Das kostet meist etwas Rechenzeit, aber man kann sich erstens darauf verlassen, dass auch wirklich alle die für diesen µC nötigen Operationen durchgeführt werden und man kann zumindest hoffen, dass die abstrakten Aufrufe auch für eine ganze Reihe ähnlicher µC gültig sind.
Diese Portabilität ist jedoch immer noch sehr eingeschränkt und sie gilt in jedem Fall nur für µC desselben Herstellers.
Eine Ausnahme bildet bei ARM-µC die Funktionssammlung für den Rechenkern, die ein Teil der Spezifikation CMSIS (Cortex Microcontroller Software Interface Standard) ist. Hier schreibt der Lizenzgeber (ARM) vor, dass bestimmte standardisierte Funktionen vorhanden

sein müssen. Für sehr viele Anwendungen genügt diese Zwischenschicht bereits, so dass die Anwendung dann direkt auf diesen Bibliotheken aufsetzt.
In komplexeren Systemen kommt jedoch typischerweise ein spezialisiertes Betriebssystem (RTOS, Real Time Operating System) zum Einsatz, das die Verwendung HW-unabhängiger Module (z.B. einem Webserver) erlaubt. Allerdings benötigt auch das RTOS natürlich Ressourcen (Speicher, Rechenleistung) und ist je nach Hersteller kostenpflichtig.

1.3 Modulbibliothek: Aufbau

Die von den Herstellern gelieferten Modulbibliotheken sind nicht untereinander austauschbar, daher kann es keine allgemeingültige Beschreibung geben. Allerdings lässt sich anhand der Bibliothek des Praktikums-µC ein recht typischer Aufbau zeigen.

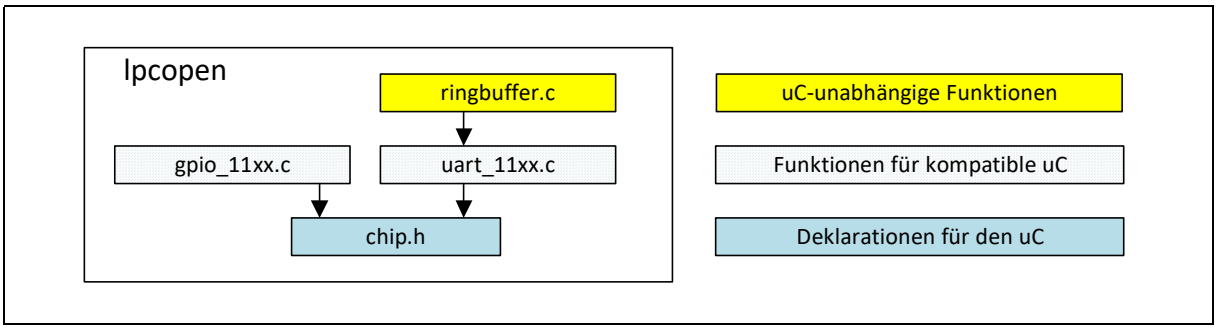

Abbildung 2: Bibliotheksmodule (Aufbau)

In diesem Fall sind sämtliche Module in einem zusammengehörigen Paket (*lpcopen*) versammelt. Zunächst gibt es für jeden µC oder Familie von µC einen Header, der spezielle Eigenschaften des µC definiert. In Abbildung 2 ist das der Header *chip.h*. Er gilt hier für die Familie LPC11xx, also auch für den im Praktikum verwendeten µC LPC11U24.

Darauf aufbauend gibt es für jedes HW-Modul eines µC eine eigene Funktionssammlung. In sind hier nur die Funktionssammlungen für zwei Module dargestellt: die GPIOs und serielle Schnittstellen (UARTs). Diese Module können aber an unterschiedlichen Adressen liegen und jeder µC kann auch eine unterschiedliche Anzahl dieser Module enthalten. Durch das Einbinden des µC-spezifischen Headers werden die Funktionssammlungen passend konfiguriert.
Der LPC11U24 enthält beispielsweise 4 Timer-Module, die über ihre Adressen auseinandergehalten werden.

In wenigen Fällen gibt es dann noch eine weitere Abstraktionsebene (hier *ring_buffer.c*). Diese Ebene ist noch abstrakter (hier: gilt prinzipiell für beliebige µC) und stellt dem Anwender häufig gewünschte Funktionen bereit, die ihrerseits dann auf eine spezielle HW zugreifen. Das Beispiel beinhaltet die effiziente Verwaltung eines vom Anwender bestimmten Speicherbereichs (buffer), mit dessen Hilfe die Kommunikation (hier: über serielle Schnittstellen) weitgehend im Hintergrund ablaufen kann.

1.4 Modulbibliothek: Anwendung

Angenommen, der Anwender möchte den 32 Bit Timer CT32B0 des µC LPC11U24 zurücksetzen, d.h. auf null stellen.
Zunächst wird im Anwenderprogramm der Header *chip.h* eingebunden. Dort findet sich zunächst die Definition der Adresse des zugehörigen HW-Moduls:

#define LPC_TIMER32_0_BASE 0x400C1000

Dies ist genau die Adresse, die man auch dem UM entnehmen könnte. Jedes HW-Modul verfügt über eigene Register, die eben an dieser Adresse beginnen. Die Menge dieser Register wird in C in einer Struktur abgebildet. Damit man ohne immer wiederkehrende Schreibarbeit gleich mit einem Zeiger auf die Struktur arbeiten kann, wird auch ein solcher Zeiger gleich definiert:

*#define LPC_TIMER32_0 ((LPC_TIMER_T *) LPC_TIMER32_0_BASE)*

Mit *LPC_TIMER32_0* kann der Anwender also direkt auf die passende Struktur verweisen.

Nun möchte der Anwender den Zähler zurücksetzen. Dazu kann er die Funktion *Chip_TIMER_Reset* aus *timer_11xx.c* aufrufen:

*void Chip_TIMER_Reset(LPC_TIMER_T *pTMR);*

Der Aufruf im Programm sähe dann so aus:

Chip_TIMER_Reset (LPC_TIMER32_0);

Sieht man sich diese Funktion in der Bibliothek selber an, dann erkennt man, dass hier doch mehr in Peripherieregistern geändert wird, als man zunächst vermuten würde.

Möchte er stattdessen den 16 Bit Timer CT16B1 verwenden, dann braucht er nur die geänderte Adresse einzutragen:

Chip_TIMER_Reset (LPC_TIMER16_1);

2 Speichernutzung

Gerade bei den sehr begrenzten Speichergrößen in µCs ist es wichtig, sich darüber klar zu werden, wo später welche Anteile des Programms im Speicher abgelegt werden und wie der zur Verfügung stehende Datenspeicher genutzt wird. Besonders wichtig ist das bei den µCs mit Harvard-Architektur. Datenspeicher ist erheblich teurer als Programmspeicher und fällt deshalb deutlich kleiner als der Programmspeicher aus (typ. Faktor 5-10).

2.1 Segmente im Programm

Der Compiler verteilt bei der Übersetzung das Programm in mehrere Segmente. Die Segmente heißen oft auch Sektionen und es kann prinzipiell beliebig viele davon geben. Allerdings sind nur drei Segmente immer vorhanden: *text*, *data* und *bss*. Sollen weitere Segmente verwendet werden, dann müssen sie im Programm mit compilerspezifischen Schlüsselwörtern angegeben werden. Zudem muss auch dem Linker dann mitgeteilt werden, wo diese zusätzlichen Segmente zu liegen kommen sollen und welche Eigenschaften sie haben sollen. Für die weiteren Erklärungen soll das nachfolgende Programm dienen.

```
1   uint16_t usage;
2   const char *meldung="Das Geraet ist jetzt betriebsbereit";
3
4   void wait(uint16_t n)
5   {
6     volatile uint16_t i;
7
8     for (i=0; i<n; i++);
9     usage++;
10  }
11
12  void main(void)
13  {
14    uint8_t led=0;
15    usage=0;
16    while (usage < 10000)
17    {
18      led ^= 1;
18      wait(1000);
19    }
20  }
```

Listing 1: Beispielprogramm

2.1.1 Segment text

In diesem Segment landet der ausführbare Maschinencode des Zielprozessors. Sie entstehen durch die Operatoren und Kontrollstrukturen des Programms. Im Beispielprogramm sind das die **nicht** markierten Anteile. Man kann bei der Übersetzung wählen, ob der Compiler möglichst kompakten Code oder möglichst schnell ausführbaren Code erzeugen soll. Wenn man nur einige wenige Programmteile möglichst schnell ausführbar machen möchte, für den großen Rest jedoch den kompakten, aber langsameren Code vorzieht, dann sollte man die Funktionen in zwei Quelldateien unterbringen. So kann man sie mit unterschiedlichen Optionen übersetzen lassen. Wie die entsprechenden Optionen dem Compiler mitzuteilen sind, muss in der jeweiligen Dokumentation nachgelesen werden. Bei dem auch im Praktikum verwendeten Compiler gcc bedeuten:

-O0: keine Optimierung (alternativ: –O ganz weglassen)
-O1: „übliche“ Optimierung (alternativ: -O)
-Os: kompakter Code

2.1.2 Segment data

In diesem Segment wird der Platz für initialisierte globale Variablen sowie initialisierte *static*-Variablen reserviert. Im Beispiel ist das die türkis markierte Meldung (ein Feld von *char*). Dabei ist zu beachten, dass diese Daten auch dann im Segment data landen, wenn sie im Programm gar nicht geändert werden oder sogar wie hier explizit als konstant (*const*) markiert werden. Die Kennzeichnung als *const* bewirkt in der Regel nur eine Prüfung bei der Übersetzung. Variablen oder Zeiger, die als *const* deklariert sind, können sich nach der Wertzuweisung nicht mehr ändern (bei *const*-Zeigern als Parametern: nach dem Funktionsaufruf). Damit kann der Compiler hier anderen (effektiveren) Code erzeugen als bei Variablen und Zeigern, die sich ändern können.
Allerdings müssen diese Variablen vor dem Programmstart die ihnen zugewiesenen Werte erhalten. Deswegen werden die Initialwerte (hier der Text der Meldung) abgetrennt und zunächst als Anhang zum Code betrachtet. Dieser Anhang wird dann beim Start des Programms noch vor dem Aufruf von *main()* an die entsprechenden Stellen im Datenspeicher kopiert. Initialisierte Variablen belegen also prinzipiell sowohl Platz im Programmspeicher als auch im Datenspeicher.

2.1.3 Segment bss

In diesem Segment wird der Platz für nicht initialisierte globale Variablen sowie nicht initialisierte static-Variablen reserviert. Vor dem Aufruf von *main()* wird dieser Bereich im Datenspeicher gelöscht (d.h. mit 0 belegt). Da keine individuelle Initialisierung erforderlich ist, wird auch kein zusätzlicher Platz im Programmspeicher benötigt. Im Beispiel landet die gelb markierte Variable *usage* im Segment bss.

2.2 Segmente im Speicher

Die in Kapitel 2.1 beschriebenen Segmente werden in der Regel nach Abbildung 3 im Speicher des µC angeordnet.

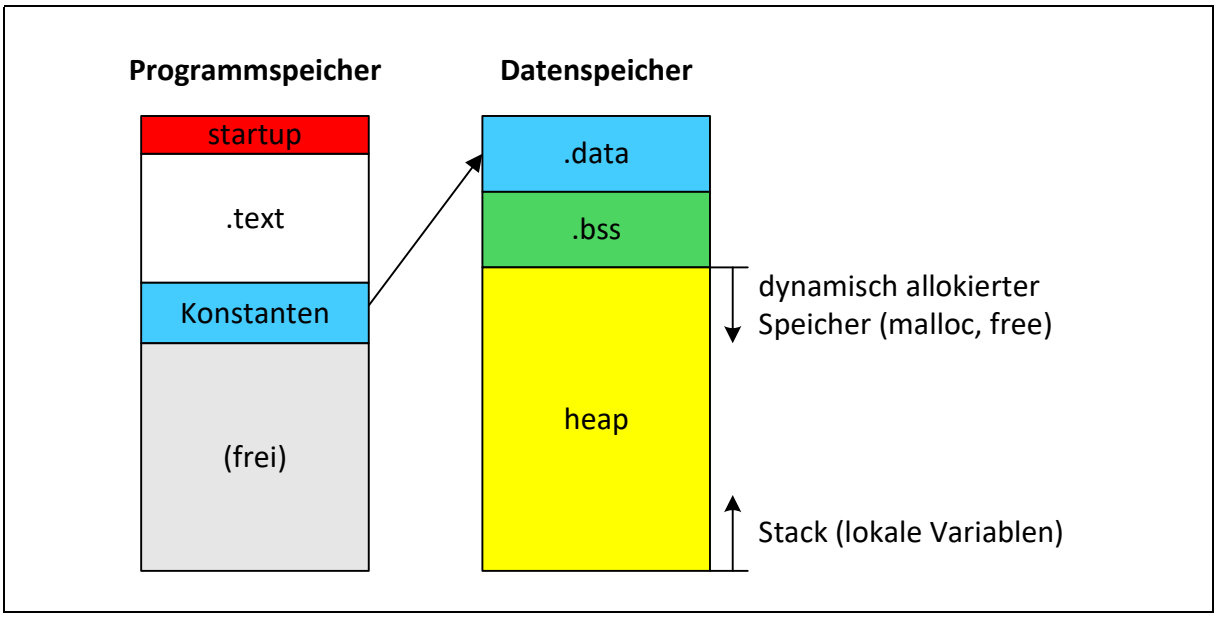

Abbildung 3: Speicherbelegung (hier: Harvard)

Zusätzlich zu den bereits bekannten Segmenten sind hier im Programmspeicher noch der Bereiche *startup* und im Datenspeicher der Bereich *heap* eingetragen.

Der Bereich *startup* enthält Code, der unmittelbar nach dem Reset aufgerufen wird. Dieser Code wird außer in Spezialfällen nicht aus dem Anwenderprogramm erzeugt, sondern wird µC-spezifisch vom µC-Hersteller oder dem Compilerhersteller mitgeliefert. Zu den Aufgaben siehe Kapitel 3.

2.2.1 Heap

Im Datenspeicher ist in der Regel der größte Bereich der *heap*, hier gelb markiert. Dies bedeutet „Haufen“ und damit ist tatsächlich der nicht fest zugeordnete „Haufen“ freien Speichers gemeint. Der Heap reicht normalerweise vom Ende des Segments bss bis zum Ende des verfügbaren Datenspeichers. Er dient zwei Zwecken.

Zum einen können Programme zur Laufzeit mittels *malloc()* dynamisch Speicher anfordern. Dieser Speicher wird dann dem Programm aus dem *heap* zugewiesen. Zu beachten ist, dass dies normalerweise eine Aufgabe des Betriebssystems ist. Wenn also (wie sehr häufig) kein Betriebssystem in einem µC-System läuft, dann muss entweder der Programmierer selbst Funktionen schreiben, die den Speicher verwalten oder er findet eine angepasste Bibliothek, die diesen Teil des fehlenden Betriebssystems nachbildet.

Der zweite Bereich ist der sogenannte *Stack* (Stapel). Dort werden funktionslokale Variablen sowie Parameter und Hilfsinformationen beim Funktionsaufruf gespeichert. Im Beispiel sind das die grün markierten Variablen und Parameter.
Zu den Hilfsinformationen gehört u.a. die Rückkehradresse. In Zeile 18 wird die Funktion *wait()* aufgerufen. Wenn diese Funktion beendet ist, soll das Programm ja in Zeile 19 fortfahren. Damit das funktioniert, wird diese Information (kehre zu Zeile 19 zurück) zur Laufzeit ebenfalls

auf dem Stack gespeichert. Damit kann eine Funktion von beliebigen Stellen im Programm aufgerufen werden, da ja die jeweilige Rückkehradresse aktuell auf dem Stack gespeichert wird. Weitere Hilfsinformationen, die auf dem Stack gespeichert werden, sind die aktuellen Werte der Arbeitsregister für den Fall, dass sie in der aufgerufenen Funktion kurzfristig für andere Zwecke benötigt werden. In dem Fall werden vor der Rückkehr zum Aufrufer die Originalwerte wieder vom Stack zurückgelesen.

Die exakte Berechnung des benötigten Speicherplatzes für den Stack ist nicht trivial. Mit jedem neuen Funktionsaufruf wird neuer Platz auf dem Stack benötigt, bei jeder Rückkehr wird der Platz wieder frei. Gefährlich ist es, große Felder als lokale Variablen zu deklarieren oder mit Rekursion zu arbeiten. Da der Platzbedarf weder für den dynamisch angeforderten Speicher noch für den Stack vorab bekannt ist, wird der Heap wie in Abbildung 3 gezeigt von beiden Enden her belegt. So ist wenigstens sichergestellt, dass der Datenspeicher vollständig ausgenutzt werden kann, bevor es zu einem Überlauf kommt. Überläufe des Stack in anderweitig benutzten Speicher führen zu schwer nachvollziehbaren Programmfehlern.

2.3 Speicheroptimierung

In einem µC-System hat man sehr häufig größere Mengen von Variablen, die eigentlich Konstanten sind. Das sind Meldungen (siehe Listing 1), aber auch Felder mit vordefinierten Werten (Beispiel: Bilder, Matrizen). Diese Daten würden alle teuren Platz im Datenspeicher benötigen. Um mit dem billigen Programmspeicher auszukommen, in dem die Initialwerte ja ohnehin schon stehen, kann man wie folgt vorgehen:

3 Startup-Code

Der Startup-Code ist ein Programmteil, der noch vor *main()* aufgerufen wird. Er ist µC-spezifisch und wird liegt fast immer schon fertig vor. Dieser Code läuft ab, noch bevor die Laufzeitumgebung für ein C-Programm vorhanden ist.
Der Code ist daher entweder von vornherein zumindest teilweise in Maschinencode geschrieben oder er ist zwar in C geschrieben, kann aber dann zunächst nicht den vollen Sprachumfang nutzen.
Daher sollte eine Anpassung auf einen neuen µC einem Experten überlassen werden. Der Startup-Code hat die folgenden Aufgaben:

1. Initialisierung des µC mit sinnvollen Startwerten (sofern nötig). Moderne µC (z.B. die typischen µC mit einem 32 Bit Kern) sind nach einem Reset gerade eben lauffähig, benötigen aber noch einige Einstellungen in Registern, um überhaupt sinnvoll eine Anwendung bearbeiten zu können.
2. Kopie der Konstanten aus dem Programmspeicher an die entsprechenden Stellen im Datenspeicher (siehe 2.1.2). und Löschen des Bereichs *bss* im Datenspeicher
3. Initialisieren des Stackpointers, d.h. Festlegen des oberen Endes des Stacksegements.
4. Aufruf der Funktion *main()*.

Da µC-Programme i.d.R. nicht beendet werden (Endlosschleife), kehrt die Funktion *main()* nicht mehr zum Startup-Code zurück. Was geschieht, wenn *main()* doch verlassen wird, muss dem jeweiligen Startup-Code entnommen werden. Gebräuchlich ist ein Halt des µC.
Der Startup-Code existiert auch in PC-Programmen und hat dort die Aufgaben 2 - 4. Der PC ist ja bereits initialisiert, so dass Aufgabe 1 entfällt. Da PC-Programme i.d.R. beendet werden, kehrt die Kontrolle dann wieder zum Startup-Code zurück. Der Startup-Code seinerseits führt eventuell noch Aufräumarbeiten aus und beendet sich (die Task) selbst durch einen passenden Betriebssystemaufruf.