1 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).

1.1 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

1.1.1 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: einfache Optimierungen
-Os: kompakter Code

1.1.2 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, soweit es den Compiler betrifft, 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.

1.1.3 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.

1.2 Segmente im Speicher

Die in Kapitel 1.1 beschriebenen Segmente werden in der Regel nach Abbildung 1 im Speicher des µC angeordnet. Hier wird angenommen, dass der µC über einen Flash-Speicher verfügt, in dem das fertige Programm abgelegt wird. Zusätzlich hat er ein RAM für die zur Programmlaufzeit benötigten Daten. Die beiden Speicher sind im Bild nur zufällig gleich groß. Die Speicheradressen wachsen von unten nach oben. Der Bereich *startup* beginnt in der Regel an der Adresse 0.

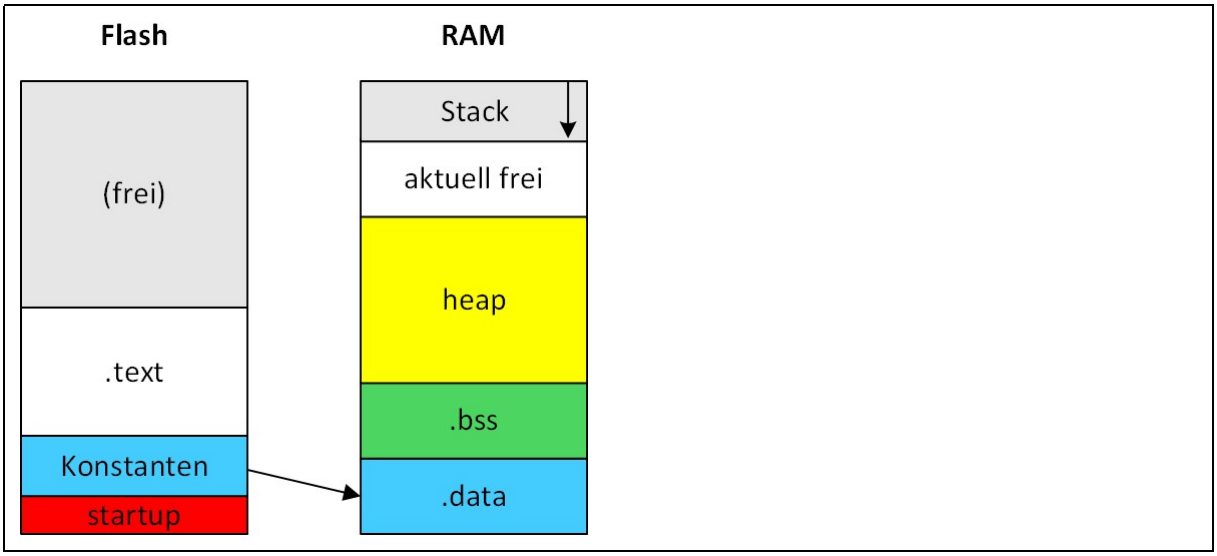

Abbildung 1: Speicherbelegung

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 uC-Hersteller oder dem Compilerhersteller mitgeliefert. Zu den Aufgaben siehe Kapitel 2.

1.2.1 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 zu einem vordefinierten Ende (d.h. einer vom Entwickler festgelegten größten Adresse im RAM). Dieses Ende darf aber nicht der größten Adresse des RAM entsprechen, weil noch Platz für den *stack* bleiben muss.

Programme können 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.

1.2.2 Stack

Dieser Speicherbereich wächst traditionell „nach unten“, d.h. die Speicherbelegung beginnt an der höchsten verfügbaren Adresse des RAM (siehe Richtungspfeil in Abbildung 1).
Hier werden lokale 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.

Die exakte Berechnung des benötigten Speicherplatzes für den Stack ist meist nicht möglich. 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. Überläufe des Stack in anderweitig benutzten Speicher (das ist hier der *heap*) führen zu schwer nachvollziehbaren Programmfehlern.

Zur Abschätzung des Speicherbedarfs für den Stack können heutige Entwurfswerkzeuge immerhin passive Hilfe leisten: Beim Start des Programms wird der Bereich oberhalb des *heap* mit einem definierten Muster (Werten) beschrieben. Dann lässt der Entwickler das Programm laufen und versucht, den worst case zu finden, d.h. einen Programmlauf mit dem vermutlich größten Speicherbedarf. Falls am Ende des Programmlaufs von dem Muster oberhalb des *heap* nichts mehr übrig ist, hat man ein Überlaufproblem.
Es gibt auch die Möglichkeit, den Compiler so einzustellen, dass vor jedem Funktionsaufruf zur Laufzeit geprüft wird, ob noch genügend Platz auf dem Stack vorhanden ist. Damit kann man zur Laufzeit ein Programm, das einen Überlauf erzeugen würde, zumindest kontrolliert beenden. Dieses Verfahren kostet natürlich zusätzliche Rechenzeit.

1.3 Speicheroptimierung

In einem µC-System hat man häufig größere Mengen von Variablen, die eigentlich Konstanten sind. Das sind Meldungen (siehe Listing 1), aber auch Felder mit vordefinierten Werten (Beispiel: Bilder). Diese Daten würden alle teuren Platz im Datenspeicher benötigen. Um mit dem billigen Programmspeicher auszukommen, in dem die Initialwerte ja ohnehin schon stehen, sollte man solche Variablen immer mit dem Schlüsselwort const als konstant deklarieren. Diese Daten werden dann zur Laufzeit in der Regel direkt aus dem Flash gelesen und belegen keinen Platz im teuren RAM.

2 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 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 1.1.2). 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.