Während Ihrer bisherigen Programmierausbildung haben Sie sich nur wenig Gedanken machen müssen, wo Ihre Variablen im Speicher liegen bzw. wie viel Speicher Sie verwenden – immerhin hatte Ihr Rechner mehrere Gigabyte zur Verfügung!
Die Hardware, die in Embedded Systems verwendet werden, bieten diesen Luxus leider nicht: Bei wenigen Kilobyte RAM und einer – im Vergleich zu Desktop-CPUs – langsamen CPU ist die Speicher- sowie Laufzeiteffizienz von größter Bedeutung.
Insofern müssen Sie als Programmierer von Mikroprozessoren auch wissen, wo Ihre Variablen im Speicher landen, welche Laufzeitimplikationen verschiedene Speichersegmente und Optimierungsstufen bringen und wie man verschiedene Optimierungsstufen geschickt einsetzt, ohne die Logik des Programms zu zerstören.

Vorbereitung

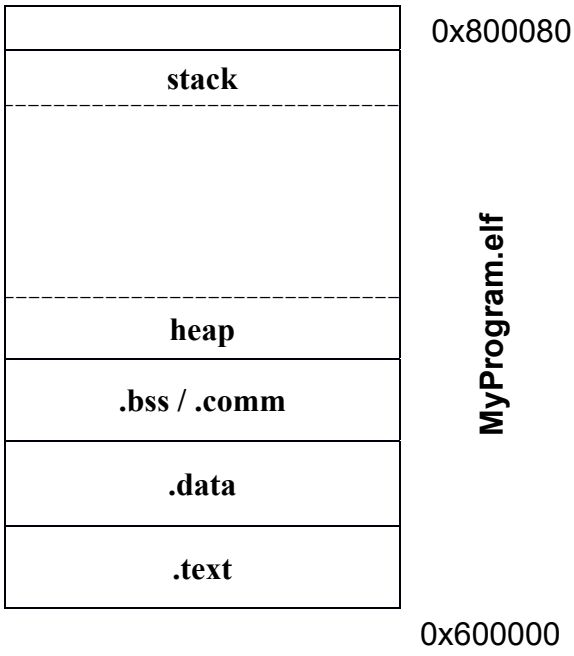
Machen Sie sich zur Wiederholung von Termin 1 nochmals mit den Funktionen der GNU C-Compiler (GCC) Toolchain vertraut, indem Sie sich die Funktion der folgenden (farblich markierten) Befehls-Flags erarbeiten.
Hierzu können Sie Online-Dokumentationen oder den Terminal-Befehl man gcc verwenden.

```
1| arm-elf-gcc -I ./inc/ -g -c FILE1.c FILE2.c FILE3.c
2| arm-elf-gcc FILE1.o File2.o File3.o -o MyProgram.elf
3| arm-elf-gcc -S FILE.c -o FILE.S
4| arm-elf-gcc -O2 FILE1.c FILE2.S -o MyFastProgram.elf
5| arm-elf-ld -Ttext 0x02000000 FILE1.o FILE2.o -o MyProgram.elf
```

Der Prefix unserer Toolchain ist **arm-elf** bzw. **arm-eb63-elf**. Dies bezeichnet die Zielarchitektur und das Binärdateiformat unserer ausführbaren Datei.
Wenn Sie den Prefix weglassen, würden Sie Maschinencode für die Host-Architektur (x86/x64) erzeugen. Da Sie jedoch keine Ausbildung in Intel-Assembly haben und unter normalen Umständen auf ein ARM-Zielsystem debuggen würden, müssen Sie beim Übersetzen Ihres Sourcecodes darauf achten, die Cross-Toolchain zu nutzen!
Neben der Ziel-Architektur ist jedoch der Aufruf an arm-elf-gcc sowie gcc identisch (d.h. die Befehle nehmen die gleichen Flags an).

1. Speichersegmente

Der benötigte Speicher eines ausführbaren Programms wird unterteilt in sog. Speichersegmente. Dies erlaubt das Aufteilen von verschiedenen Datenzugriffen auf versch. Speicherräume und isoliert die versch. Daten (semantisch) voneinander.
Während es möglich ist, benutzerdefinierte Speichersegmente zu erstellen, findet man in der Literatur für gewöhnlich folgende Segmentierung:



a) Sie waren aus der C++-Programmierung gewohnt, das **heap**-Segment zu nutzen, um zur Laufzeit neuen Speicher zu allokieren.
Ist dies bei der Programmierung für Embedded Systemen ebenfalls gebräuchlich?

b) Untersuchen Sie das Speicherverhalten von lokalen Variablen (also jene, die innerhalb von geschweiften Klammern deklariert werden).
Überführen Sie hierzu die beigelegte C-Datei in eine Assembler-Datei und beantworten Sie die Fragen im Fragenkatalog.

c) Nehmen Sie an, dass Sie eine rekursive Funktion haben, die viele lokale Variablen deklariert. Diese Funktion hat ein Wachstum von $O(3n * n^4)$ und muss eine große Liste ($n > 1000$) an Werten abarbeiten.
Welche Probleme könnten bzgl. der Speichersegmente auftreten?

d) Untersuchen Sie das Speicherverhalten von globalen Daten (also jene, die außerhalb von Funktionen deklariert werden). Man unterscheidet zwischen initialisierte und nicht-initialisierte globalen Daten!
Überführen Sie hierzu die beigelegte C-Datei in eine Assembler-Datei und beantworten Sie die Fragen im Fragenkatalog.

e) In welchem Speichersegment befindet sich Ihr Code?

f) Verwenden Sie auch den Befehl arm-elf-nm, um die Symboltabelle der ELF-Datei auszugeben. Um die Ausgabe des nm-Befehls zu verstehen, können Sie das Manual durchlesen mit dem Befehl man nm.
Welche Symbole können Sie Ihrem Programm zuordnen? Welche weiteren Symbole gibt es? Welche Bedeutung könnten die weiteren Symbole haben?

Beantworten Sie die Fragen zu den jeweiligen Aufgaben im Fragenkatalog!

2. Optimierungsstufen

Während wir in der Assembler-Sprache die Möglichkeit hatten, unsere Funktionen kompakt und effizient zu programmieren, fehlt uns in der Hochsprache diese Möglichkeit: Uns ist es nicht möglich, Entscheidungen bzgl. der genutzten Register, Speicherzugriffe, Programmsprünge usw. direkt zu beeinflussen.

Bei der Bearbeitung der obigen Aufgaben ist Ihnen bestimmt schon aufgefallen, dass aus einer einzigen C-Codezeile mehrere Assembler-Zeilen generiert werden. Der generierte Code ist zudem ineffizient und erscheint unnötig kompliziert. Dies liegt an der Arbeitsweise des Compilers: Jede C-Code-Anweisung wird durch eine fest eingebaute Assembler-Code-Folge ersetzt, um eine effektive und stabile Codeerzeugung zu gewährleisten.

Durch die Optimierungsstufen können wir jedoch Einfluss nehmen auf die Erstellung der Assembler-Anweisungen, um das resultierende Programm auf verschiedene Anwendungsanforderungen anzupassen. So kann aus dem gleichen, unveränderten C-Code verschiedene Assembler-Outputs erzeugt werden, die verschiedene *Optimierungsziele* verfolgen.
Machen Sie sich mit den verschiedenen Optimierungsstufen des GNU C-Compilers vertraut. Untersuchen Sie hierfür die Optimierungsstufen: {'0‘, '1‘, '2‘, 's‘}

a) Welche Optimierungsstufe würden Sie wählen, wenn Sie die Laufzeit Ihres Programms verkürzen möchten?

b) Welche Optimierungsstufe würden Sie wählen, um die Codegröße (und damit die resultierende Datei) möglichst klein zu halten?

c) Kompilieren Sie die beigelegte C-Datei mit den obigen Optimierungsstufen und erstellen eine ausführbare Datei pro Optimierungsstufe:

arm-elf-gcc –O0 Aufgabe2_c.c –o Aufgabe2_c_O0.elf

arm-elf-gcc –O1 Aufgabe2_c.c –o Aufgabe2_c_O1.elf
…

Führen Sie nun für jede ELF-Datei folgenden Befehl aus:
time arm-elf-run Aufgabe2_c_O{OPTIMIERUNGSSTUFE}.elf

Dieser Befehl führt Ihren Code aus und zeigt Ihnen daraufhin an, wie viel Zeit die Ausführung des Programms benötigt hat. Machen Sie mehrere Zeitmessungen pro Optimierungsstufe und ziehen Sie den Durchschnitt daraus, um etwaige Ausreißer zu glätten.

d) Untersuchen Sie das Ergebnis der Codeoptimierung für die beigelegte Datei. Überführen Sie hierzu die C-Datei in zwei Assembler-Dateien mit jeweils der Optimierungsstufe 0 und 1.

e) Untersuchen Sie das Ergebnis der Codeoptimierung für die beigelegte Datei. Überführen Sie hierzu die C-Datei in drei Assembler-Dateien mit jeweils der Optimierungsstufe '1', '2' und 's'.

Beantworten Sie die Fragen zu den jeweiligen Aufgaben im Fragenkatalog!

3. Meilenstein

Sie haben den beiliegenden Fragenkatalog ausgefüllt. Die bei der Beantwortung der Fragen entstandenen Unklarheiten konnten Sie klären.