Präprozessor, Compiler, Linker

Jörn Loviscach

Versionsstand: 7. Oktober 2011, 11:24

Die nummerierten Felder sind absichtlich leer, zum Ausfüllen in der Vorlesung.
Videos dazu: http://www.j3L7h.de/videos.html

1 Übersetzung, Interpreter, Compiler

Praktisch kein Mikroprozessor versteht direkt die Sprache C. Die verschiedenen Familien an Mikroprozessoren (z. B. x86 und x64 für die in PCs üblichen Prozessoren) haben alle ihre eigenen, recht simplen Maschinensprachen. Diese Binärcodes fasst man nur selten an – und wenn, dann nur in der Form von Assemblercode, in dem die binären Muster durch halbwegs lesbare Symbole ersetzt sind. Demo: MSP430-Assembler.

Also müssen die höheren Sprachen wie C aus dem „Quellcode“ [source code] in die jeweilige Maschinensprache übersetzt werden. Dafür gibt es zwei grundsätzliche Strategien:

[1]

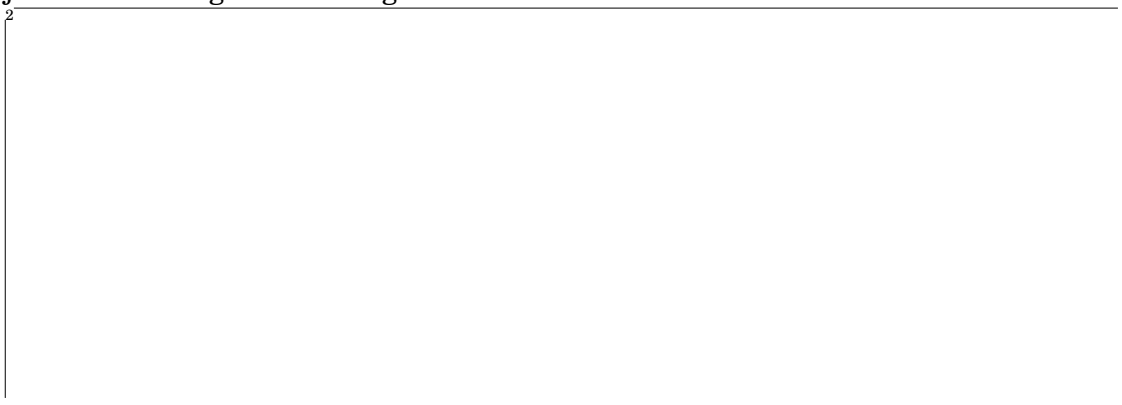
Skriptsprachen sind typischerweise von der ersten Sorte, „vollwertige“ Programmiersprachen von der zweiten.

Diese harte Unterscheidung ist aber längst veraltet: Skriptsprachen wie JavaScript werden zur Beschleunigung bei der Ausführung in Maschinensprache gewandelt (Just-in-Time Compiler). Die üblichen Compiler für die „vollwertigen“ Programmiersprachen Java und C# erzeugen keine Maschinensprache (native code), sondern die Sprache einer virtuellen Maschine (Bytecode bzw. Intermediate Language). Diese Sprache wird interpretiert (wieder mit einem Just-in-Time Compiler) bzw. beim ersten Programmstart in Maschinensprache übersetzt.

Anmerkung am Rande: Alle gängigen Programmiersprachen verwenden für den Quellcode normale Textdateien. Die Namen der Dateien enden dann zwar auf .c, .cpp, .java, .cs, .js usw., aber man kann die Dateien trotzdem mit einem Texeditor öffnen. Nicht dagegen den kompilierten Code, zum Beispiele eine .exe-Datei unter Windows! (Demo)

2 Übersetzungsvorgang in C und C++

In den Sprachen C und C++ bekommt man anders als in modernen Sprachen viel vom Übersetzungsvorgang mit – meist mehr, als einem lieb ist. In C und C++ hat der Übersetzungsvorgang drei Schritte. Dafür ruft die Entwicklungsgebung jeweils ein einiges Werkzeug auf:



In den üblichen Entwicklungsumgebungen wie der IAR Embedded Workbench oder Microsoft Visual Studio passiert alles drei beim Klick auf Run, Debug, Make, Rebuild oder Erstellen automatisch hintereinander.

Warum diese Arbeitsteilung? Der Präprozessor sah früher mal wie eine gute Idee aus, um Programmcode automatisch umzuformen. In Java und anderen modernen Vertretern der C-Familie fehlt er ganz. Die Idee hinter der Trennung von Compiler und Linker war, dass man nur die Teile des Programms übersetzen muss, die man gegenüber dem letzten Compilerlauf geändert hat. Für die anderen wird der bestehende Code genommen. Der Linker bindet diese (Objekt-)Dateien dann recht schnell zusammen. (Demo)

Die erste Bekanntschaft mit den dreien macht man durch Fehlermeldungen (errors): Eine Zeile wie `int = 4;` gibt eine Fehlermeldung vom Compiler; deklariert man eine Funktion und verwendet sie, definiert sie aber nicht, gibt das eine Fehlermeldung vom Linker. (Demo)

Bei Fehlermeldungen wird *kein* ausführbares Programm erzeugt! Gibt es bei der Übersetzung nur Warnungen (warnings) statt Fehler, wird ein ausführbares Programm erzeugt – allerdings wahrscheinlich eines, das nicht tut, was es soll.

3 Präprozessor

Der Präprozessor von C und C++ hat eigene Befehle, die alle mit einem Doppelkreuz # beginnen und *nicht* mit einem Semikolon abgeschlossen werden.

Der wichtigste Befehl des Präprozessors ist `#include`. Hier wird der Inhalt einer Datei eingelesen und exakt an dieser Steller plaziert, so als ob man sie hineinkopiert hätte. So liest `#include "servo01.h"` die Datei servo01.h ein und tut so, als ob deren Inhalt an dieser Stelle stünde. Dies ist die übliche Methode, Funktionsdeklarationen zu verteilen: Man schreibt sie in eine eigene Datei und bindet sie mit `#include` ein. Solche Dateien heißen Header-Dateien, weil sie am Kopf der C-Dateien stehen, daher auch das Kürzel .h.

`#include` gibt es in zwei Varianten: `#include "..."` liest eine Datei ein, die unter anderem neben der aktuellen C-Datei stehen kann. `#include <...>` liest eine Datei ein, die systemweit in einem eigenen Verzeichnis steht. Dort finden sich zum Beispiel allgemeine Header-Dateien wie stdbool.h – die Datei, in der unter anderem `true` und `false` definiert werden.

Kann die „inkludierte“ Datei nicht gefunden werden (Tippfehler beim Namen?), bricht der Compiler mit einer Fehlermeldung ab – nicht der Präprozessor, denn der Präprozessor ist in den Compiler integriert oder wird von dem Compiler zu Beginn aufgerufen. (Demo)

Der zweitwichtigste Befehl des Präprozessors ist `#define`. Dies ist ein automatisches Suchen und Ersetzen. Zum Beispiel `#define NUMBER_OF_CHANNELS 16` sagt, dass die (traditionell in Großbuchstaben gesetzte) Zeichenfolge `NUMBER_OF_CHANNELS` überall durch die Zeichenfolge `16` ersetzt werden soll. Das ist eine von vielen Möglichkeiten, benannte Konstanten zu erzeugen (vgl. `BIT0` bis `BITF` in io430G2231.h). Das ist heute verpönt. In C99 und C++ schreibt man sicherer ohne Präprozessor: `static const int NumberOfChannels = 1234;`

Man kann obendrein Werte an ein „Präprozessor-Makro“ übergeben: `#define SQUARE(x) x*x` ersetzt zum Beispiel `int a = SQUARE(b);` durch `int a = b*b;`. Das passiert wohlgemerkt vor dem Kompilieren. Diese Makros sind aber trügerisch und deshalb ebenfalls verpönt: Was macht `int c = SQUARE(d+e);`?

[3]

In C99 schreibt man sicherer ohne Präprozessor:
`static inline int square(int x) {return x*x;}`

Im Endeffekt muss man `#define` praktisch nur noch an einer Stelle verwenden: Als „`#include` Guard“ um eine Header-Datei wie in servo01.h:

[4]

Diese drei Präprozessor-Befehle stellen sicher, dass diese Header-Datei pro C-Datei nur einmal einkompiliert wird – sonst bekommt man Probleme bei komplexeren Programmen mit vielen Header-Dateien. Dieser Schutz gegen das zweimalige Kompilieren funktioniert so:

[5]

Der Präprozessor ist nicht nur eine relativ dumme Maschinerie – man kann Befehle an ihn auch nur schwierig debuggen. Für den Präprozessor gibt es keinen Einzelschritt-Debugger; immerhin kann man sich meist das Ergebnis nach dem Präprozessor ansehen. (Demo) Dass der C-Code durch den Präprozessor verändert wird, erschwert obendrein das normale Debuggen.

Also wird man den Präprozessor heute möglichst nur noch für zwei Sachen benutzen:

[6]

4 Compiler

Der Compiler erledigt die Hauptaufgabe der Übersetzung. Er liest, was der Präprozessor aus einer C- oder C++-Datei gemacht hat, und produziert daraus eine Objektdatei. Die enthält Maschinensprache und Angaben, die der Linker benötigt, um die verschiedenen Funktionen aus den einzelnen Objektdateien zu verknüpfen.

Im Projekt in der Entwicklungsumgebung gibt man nur an, welche C- oder C++-Dateien übersetzt werden sollen, aber nicht, welche Header-Dateien dazu gehören. Die C- oder C++-Dateien fordern die Header-Dateien ja selbst per `#include` an.

Der Compiler liefert gerne und viele Fehlermeldungen – zum Beispiel, wenn man Variablen verwendet, die man nicht vorher eingeführt hat (Tippfehler?) oder die an der aktuellen Stelle unsichtbar sind, oder wenn man ein Semikolon vergisst. Achtung: Der Fehler tritt vielleicht erst eine Zeile später auf. (Demo)

Ebenso liefert der Compiler gerne und viele Warnungen. Die nerven am Anfang, aber man lernt sie schnell zu schätzen. Ein gut geschriebenes Programm wird ohne Warnungen kompiliert. Zum Beispiel `if(i=13){...}` führt zu einer Warnung (Demo). Ebenso gibt der Compiler Warnungen, wenn eine Variable berechnet wird, aber der Wert nicht weiterverwendet wird, und wenn Befehle wegen `break;` oder `return ...;` nie erreicht werden können. (Demo)

5 Linker

Der Linker bindet die Objektdateien, die der Compiler erzeugt hat, und gegebenenfalls noch „statische“ Funktionsbibliotheken zu einem ausführbaren Programm zusammen. An dieser Stelle kann nicht mehr allzu viel schief gehen.

Der typische Linker-Fehler ist, dass eine Funktion, die man verwendet, nicht definiert ist – weil man sie schlichtweg vergessen hat oder weil man vergessen hat, die entsprechende C-Datei in das Projekt einzubinden. (Demo)

Seltener kann es auch passieren, dass man die Funktion gleich zweimal definiert hat. Das findet schon der Compiler, wenn beide Definitionen beim Compilieren einer einzigen C-Datei auftauchen. Sonst findet es der Linker.

So könnte sich die Funktion namens `initialize` in einer Variante auf Servos und in einer anderen Variante auf Sensoren beziehen. Das kann hakelig werden, wenn man mehrere fremde Bibliotheken benutzt. Ab C++ aufwärts gibt es dafür saubere Lösungen.

Anders als in C++ und seinen Nachfolgern darf es in C99 keine zwei Funktionen mit gleichem Namen geben, auch wenn man ihnen jeweils andere Typen an Parametern gibt: C99 kennt kein „Überladen“ von Funktionen.

Der Linker meldet – anders als der Compiler – Fehler, ohne die Zeilennummern anzugeben. Das macht die Fehlersuche aufwändiger.

6 Sichtbarkeit auf Ebene der Dateien

Mit `static` und `extern` kann man in C und C++ steuern, ob Funktionen und globale Variablen, die in einer C- oder C++-Datei definiert sind, auch in allen anderen Dateien sichtbar werden können. Leider sind diese beiden Namen verwirrend; in moderneren Sprachen ist dieses Gestrüpp aufgeräumt.

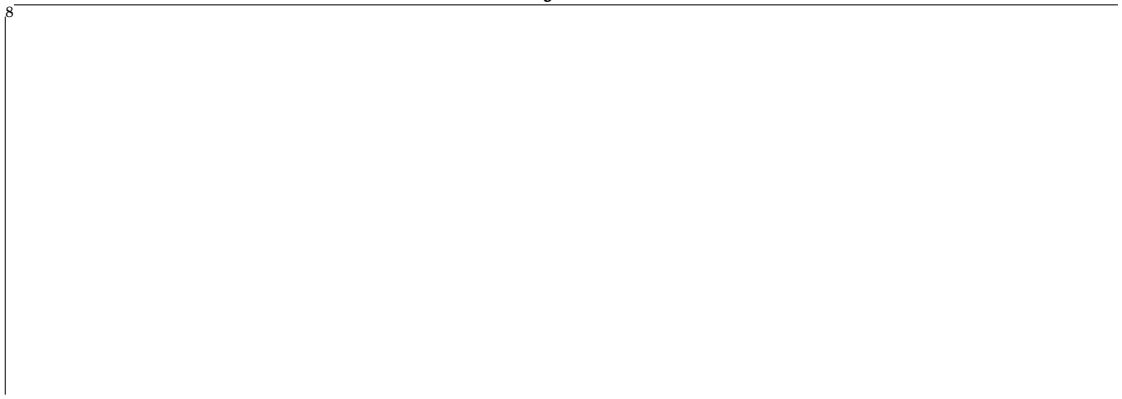
Ein Warnhinweis vorab: Globale Variablen sind zu vermeiden. Je enger die Sichtbarkeitsbereiche von Variablen sind, desto weniger Unsinn kann man mit ihnen anstellen und desto weniger Konflikte gibt es. Mehr dazu später.

In einer C-Datei könnte dieses stehen:

[7]

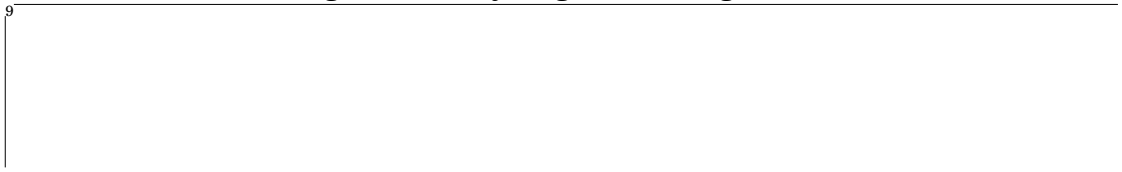
In einer anderen C-Datei desselben Projekts könnte dieses stehen:

[8]



Dann werden über das gesamte Projekt gemeinsam genutzt:

[9]



Und jede der beiden C-Dateien hat diese jeweils für sich:

[10]

Um die Werte der `static`-Variablen zu verfolgen, muss man in der IAR Embedded Workbench beim Debuggen noch View > Statics öffnen.

Zur Erinnerung: Innerhalb einer Funktion bedeutet `static int a = 42;` etwas ziemlich Anderes, nämlich dass der Wert von `a` ganz zu Beginn auf 42 gesetzt wird, aber dann von einem Aufruf der Funktion zu ihrem nächsten Aufruf überlebt.