WHITEPAPER

On- und Off-Target Ausführung im Vergleich

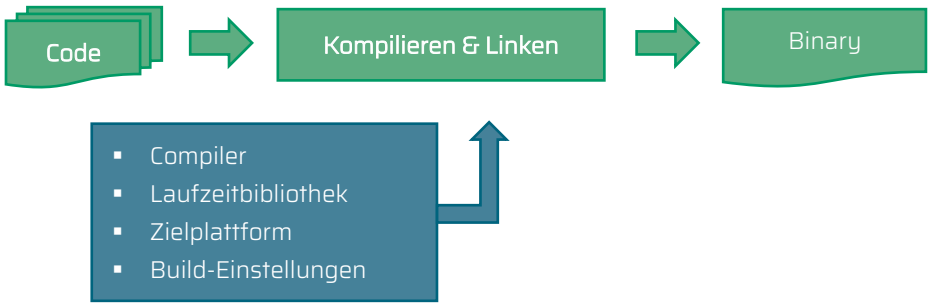
Embedded Firmware wird üblicherweise auf einem Desktop-PC entwickelt.
Code ohne direkte Abhängigkeit zur Hardware kann mit einem herkömmlichen Desktop-Compiler kompiliert und bereits auf dem Entwicklungsrechner (*Off-Target*) ausgeführt werden. Dieser Off-Target Ansatz wird häufig gewählt, um lange Kompilieren-Flashen-Debug Schleifen auf dem Mikrocontroller zu umgehen.[1]

Dabei muss allerdings beachtet werden, dass der so getestete Code bei der Ausführung auf dem echten Zielsystem (*On-Target*) unter Umständen ein anderes Verhalten zeigen kann. Im besten Fall kommt es bereits bei der Kompilierung mit dem Cross-Compiler zu einem Fehler. Mit weniger Glück verhält sich der kompilierte Code in Details anders. Im Off-Target Verfahren „verifizierter" Code ist plötzlich nicht mehr korrekt.

Vielfach wird der Off-Target Ansatz für die Ausführung von Unit-Tests gewählt. Das ist im Hinblick auf ein effizientes Entwickeln sehr sinnvoll. Werden diese Tests allerdings nicht zusätzlich auch On-Target – mit potentiell unterschiedlichen Ergebnissen! – ausgeführt, ergibt sich schnell eine rein „gefühlte Sicherheit".

Dieses Whitepaper demonstriert die wichtigsten technischen Gründe für Unterschiede zwischen On- und Off-Target Ansatz.

Unterschiede bei der Erzeugung



Compiler und Laufzeitbibliothek

Verschiedene Compiler haben unterschiedliche **Unterstützung für Features**. Dies wird besonders dann zum Problem, wenn Sprachfeatures aus vergleichsweisen aktuellen Standards (beispielsweise C18 und C++17) eingesetzt werden. Die meist verwendeten Desktop-Compiler (Visual Studio, gcc, clang) unterstützen nahezu immer deutlich mehr

[1] James W. Grenning: Test-Driven Development for Embedded. The Pragmatic Programmers, 2011, S. 77f.

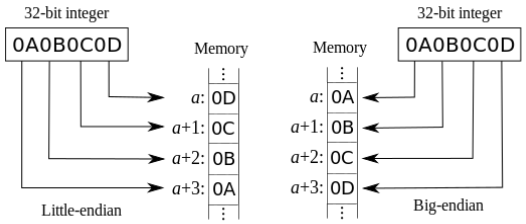
Features als ein Cross-Compiler gleichen Datums. Code, der solche Features verwendet, wird für die Zielplattform nicht kompilieren und muss umständlich umgeschrieben werden.

Compiler und Laufzeitbibliotheken haben **Bugs**. Diese Tatsache mag zunächst erstaunen, ergibt sich jedoch naturgemäß aus der mittlerweile enormen Komplexität dieser Tools. Das tatsächliche Ausmaß dieser Fehlerquelle ist dramatisch. [2]
An einem konkreten Bug in der GNU Embedded Toolchain von ARM werden die Auswirkungen deutlich: Bugticket #1527413 „*4.9 series reproducibly corrupts register R7*“.[3] Unter sehr speziellen Bedingungen wird hier ein Prozessor Register ungewollt verändert. Der Beispielcode zur Reproduktion würde unverändert mit einem anderen Compiler erfolgreich ausführen. Damit ein Fehler dieser Art entdeckt werden kann, muss der Code On-Target ausgeführt werden.

Zielplattform

Die Größe von Datentypen in C und C++ ist implementierungs-abhängig. So ist ein Zeiger auf dem Entwicklungs-PC heute 8 Byte groß. Auf einem 32-bit Zielsystem ist ein Zeiger dagegen üblicherweise 4 Byte groß. Gleiches gilt für den long Datentyp. Hinzu kommen das sogenannte Alignment. Prozessoren benötigen teilweise beim Lesen/Schreiben eines Wortes eine Speicheradresse, die mit der Wortgröße ausgerichtet ist. Der Compiler kennt diese Anforderungen und ordnet Werte entsprechend an.

Datentyp-Größe und Alignment führen dazu, dass Objekte von Strukturen oder Klassen eine unterschiedliche Größe und Repräsentation im Speicher erfahren. Hinzu kommt die unterschiedliche Interpretation von Wörtern als *Little*- oder *Big-Endian*.



Gerade in Low-Level Embedded Code wird Speicher vielfach byte-weise interpretiert, kopiert und übertragen. Der eingesetzte Code ist durch die dargestellten Eigenschaften schnell unwisssentlich von der Zielplattform abhängig.

Build-Einstellungen und Optimierung

Sprachstandards wie C und C++ enthalten **undefiniertes Verhalten**.[4]

---

[2] Der populäre gcc Compiler weist Stand Januar 2020 in seinem Bug-Tracker 13.495 offene Bugs aus. In der vergangenen Woche konnten 72 davon behoben werden. Gleichzeitig wurden 81 neue Bugs entdeckt.
[3] https://bugs.launchpad.net/gcc-arm-embedded/+bug/1527413
[4] https://en.cppreference.com/w/cpp/language/ub

Enthält ein Programm Code, der nach dem Standard nicht definiert ist, kann der Compiler diesen willkürlich behandeln. Compiler machen von dieser zulässigen Möglichkeit gerade im Zusammenhang mit Optimierung Gebrauch.

```
int f(bool parameter) {
      int a;    // uninitialisierte lokale Variable
      if(parameter) { a = 42; }
      return a; // potentiell Zugriff auf unininitialisierte Variable
}
```

Der gezeigte Code ist undefiniert, weil potentiell ein Zugriff auf eine uninitialisierte Variable durchgeführt wird. Die Effekte werden anhand des Assembly-Outputs [5] eines arm gcc 8 Compiler analysiert:

- Bei ausgeschalteter Optimierung wird der Übergabeparameter auf Ungleichheit mit 0 geprüft und dann 42 zurückgegeben. Andernfalls wird der Wert der lokalen Variable a zurückgegeben. Für diese wurde Speicher auf dem Stack reserviert, aber nicht initialisiert. Der Rückgabewert ist abhängig davon, was vorher in dieser Stelle RAM gespeichert wurde.
- Bei eingeschalteter Optimierung wird der Parameter-Wert nicht ausgewertet und immer 42 zurückgegeben. Das vorhandene undefinierte Verhalten erlaubt dem Compiler diese drastische – und wenig intuitive! - Abkürzung.

Die Optimierungs-Stufe kann also deutlichen Einfluss auf die Ergebnisse eines Programmes haben. Die Einstellungen für das Kompilieren & Linken müssen bei der Verifizierung unbedingt beachtet werden.

Unterschiede bei der Ausführung

Im vorigen Abschnitt wurde gezeigt, welche Unterschiede im Hinblick auf On- und Off-Target Ansatz bereits bei der Erzeugung eines Binaries bestehen. In diesem Abschnitt werden kurz die Unterschiede bei der Ausführung des Binaries beleuchtet.

Laufzeitumgebung

Die zur Laufzeit vorhandene Umgebung auf dem Mikrocontroller unterscheidet sich in vielen Punkten deutlich von der auf dem Desktop-PC. Manche Eigenheiten sind offensichtlich, so bspw. der begrenzte RAM und damit auch Stack auf dem Zielsystem. Andere Punkte werden erst bei genauer Betrachtung deutlich. Falls auf dem Zielsystem bspw. überhaupt ein Heap zur Verfügung steht, wird dieser intern andere Charakteristiken (Allokations-Strategie und Fragmentierung) aufweisen.

---

[5] https://godbolt.org/z/TpK2d-

Wird ein Echtzeitbetriebssystem eingesetzt, bietet dieses u. U. eine mit dem Desktop-PC kompatible Portierung an. Jedoch werden dennoch alle Betriebssystem-Aufrufe hinter der gemeinsamen API eine deutlich abweichende Implementierung aufweisen.

Hardware

Beim Off-Target Ansatz wird das Binary auf der Hardware eines Desktop-PC ausgeführt. Die verarbeitende CPU unterscheidet sich natürlich deutlich von dem Mikrocontroller der Zielplattform.

So wie bereits bei Compilern gezeigt, können auch Mikrocontroller Fehler enthalten. Für nahezu jede MCU existiert ein Erratum, das solche Fehler auflistet. Im Großteil der Fälle handelt es sich hier allerdings um Fehler in der Peripherie des Mikrocontrollers. Besonders komplexe Peripherie-Schaltungen wie bspw. ein DMA-Controller sind davon betroffen. Eine korrekte Ausführung bezogen auf solche Fehler kann prinzipiell nur On-Target erfolgen, da diese Peripherie, und dementsprechend auch der zugehörige Code, plattformspezifisch ist.

Obwohl deutlich seltener, gibt es durchaus auch strukturelle Fehler im Rechenkern eines Mikrocontrollers. Beispielhaft soll hier ein Problem beim ARM Cortex M4F genannt werden: FPU Errata 1299509 "*Fused MAC instructions give incorrect results for rare data combinations*" [6]

Falls für die Zielplattform ein Simulator existiert, kann dieser für die Ausführung verwendet werden. Allerdings können Simulatoren selbst andere Fehler haben oder die auf der realen Hardware vorhandenen Fehler nicht korrekt nachbilden. [7]

Fazit

Entwickler führen Code auf dem Entwicklungsrechner aus, um die mit der On-Target Ausführung verbundenen langen Feedbackschleifen (Kompilieren-Flashen-Debug) abzukürzen. Dieses Whitepaper hat die wichtigsten Gründe für die strukturellen Unterschiede zwischen einer *On-* und *Off-Target* Ausführung aufgezeigt. Verschiedene Einflussfaktoren können dazu führen, dass im *Off-Target* Verfahren „verifizierter" Code auf dem Zielsystem plötzlich nicht mehr korrekt arbeitet.

Ein solides Verständnis dieser Auswirkungen ist zur Fehlervermeidung unbedingt erforderlich. Praktisch sollte dennoch in jedem Fall mindestens periodisch eine Verifizierung auf dem Zielsystem stattfinden.

---

[6] https://static.docs.arm.com/epm039104/30/Cortex-M4_Software_Developers_Errata_Notice_v3.pdf

[7] Stephan Grünfelder: Software-Test für Embedded Systems. dpunkt.verlag, 2013, S. 109f.