Coding Rules für Embedded Software

Carsten Steul

19. Juni 2011

Inhaltsverzeichnis

1 Coding Rules

1.1 Was sind Coding Rules?

Coding Rules sind Richtlinien zur Programmierung. In diesen ist festgelegt wie Code geschrieben werden soll, aber auch wie Code nicht geschrieben werden soll. Diese Richtlinien sind oft in Standards zusammengefasst. [JSF AV, S. 7]

1.2 Warum werden Coding Rules benutzt?

Sie dienen dazu bestimmte Code-Konstrukte zu verbieten oder zu fördern um bekannte und/oder häufige Fehler zu vermeiden. Des Weiteren sind viele davon darauf bedacht Code portabel zu halten und Plattformeigenheiten zu umgehen. Auch soll der Code lesbar und wartbar sein. Es gibt auch Designrichtlinien welche vor allem die Abhängigkeiten zwischen Komponenten verringern sollen, was sich auch wieder auf die Wartbarkeit auswirkt. [JSF AV, S. 7,9,13][CERT, Section 00][Stroustrup]

1.3 Wie sollten Coding Rules ausgewählt werden?

Richtlinien sollten am besten automatisch überprüfbar sein, denn so können sie z.B. bei jedem Compiliervorgang überprüft werden. Je nach Projekt empfiehlt es sich Richtlinien auszuwählen die zu dem Projekt passen. Vor allem bei sicherheitskritischen Systemen sollte die Auswahl gut geplant werden und am besten auf vorhandene und vielfach genutzte Standards zurückgegriffen werden. [Misra Website][CERT, Section 00]

2 Standards

Es gibt viele Standards, doch die meisten sind nicht weit verbreitet da es Eigenentwicklungen für bestimmte Projekte sind. Zu den verbreitetsten Zählen MISRA, JSF AV und der CERT Secure Coding Standard. Meistens werden diese aber für das jeweilige Projekt angepasst oder kombiniert. Die meisten Richtlinien die in diesen Standards enthalten sind beziehen sich nicht auf Embedded Software sondern sind allgemein gehalten. So können diese Standards durchaus auch außerhalb der Embedded Entwicklung angewendet werden. [Misra Website][JSF AV][CERT]

2.1 MISRA-C

MISRA (Motor Industry Software Reliability Association) wurde von Automobilherstellern und Zulieferern entwickelt und 1998 veröffentlicht. Er sollte dazu dienen C-Code zu vereinheitlichen und Fehler zu vermeiden. Der Standard wird mittlerweile auch in anderen Industrien benutzt und ist dadurch weit verbreitet. Unregelmäßig wird der Standard aktualisiert und angepasst. Auch eine spezielle C++ Variante existiert bereits. Viele IDEs für Embedded Software bringen Programme mit die den Code nach den MISRA Regeln überprüfen. Ein Nachteil ist, das der Standard kostenpflichtig ist und damit für einige mögliche Anwender den finanziellen Rahmen sprengt. [Misra Website][MISRA]

2.2 JSF AV

Der JSF AV(Joint Strike Fighter Air Vehicle) wurde von Lockheed Martin in Zusammenarbeit mit Liverpool Data Research Associates für den Joint Strike Fighter entwickelt und 2005 veröffentlicht. Dieser ist speziell auf C++ und Luftfahrzeuge zugeschnitten, lässt sich aber auch auf andere Bereiche anwenden. Da auf MISRA aufgebaut wird sind fast alle MISRA Regeln auch in JSF AV enthalten, wurden aber wenn es nötig war angepasst oder weggelassen. Bjarne Stroustrup, der C++ Erfinder, empfiehlt diesen Standard bei der Entwicklung von Embedded Software mit C++ gegenüber proprietären C++ Dialekten wie EC++.[Stroustrup][EC++] Der Standard ist kostenlos im Internet verfügbar, allerdings sind wenig Beispiele enthalten. [JSF AV]

2.3 CERT Secure Coding Standard

Der CERT Secure Coding Standard wird vom CERT zusammen mit freiwilligen mit Hilfe eines Wikis entwickelt. Er enthält im Gegensatz zu MISRA und JSF AV allgemeine Richtlinien, die sich aber auch auf Embedded Software anwenden lassen. Neben den Richtlinien für C gibt es auch welche für C++ und Java. Die für Java wurden in Zusammenarbeit mit Sun/Oracle entwickelt. Der Standard ist ebenfalls kostenlos und enthält im Gegensatz zu den anderen viele Beispiele. [CERT]

3 Beispiele für Richtlinien

Die Standards haben teilweise hunderte Richtlinien. In den Beispielen beschränke ich mich auf die für Embedded Software wichtigsten und die, die die häufigsten Fehler vermeiden.

3.1 Basistypen

Die Basistypen, char, short, int, long, float und double besitzen je nach Plattform und dort manchmal auch je nach Compiler eine andere Größe. Deswegen sollten immer Typen mit festgelegter Größe verwendet werden, wie zum Beispiel int8_t, also ein signed int der genau 8 Bit hat.

Listing 1: Falsch

```
int a;
a = 2000000;
print(a); //2000000 wird nur ausgegeben falls int 32 Bit oder mehr hat
```

Listing 2: Richtig

```
int32_t a;
a = 2000000;
print(a); //2000000 wird ausgegeben
```

[MISRA, Rule 13][JSF AV, Rule 209][CERT, INT00-C]

3.2 Blöcke

Blöcke sollten immer mit {} umgeben werden um Fehler zu vermeiden. Diese Fehler treten vor allem auf wenn der Code geändert wird und der Programmierer übersieht, dass die Klammern fehlen.

Listing 3: Falsch

```
for(i = 0; i < 100; ++i)
    Do();

for(i = 0; i < 100; ++i)
    Do();
    Do2();
```

Listing 4: Richtig

```
for(i = 0; i < 100; ++i) {
    Do();
}

for(i = 0; i < 100; ++i) {
    Do();
    Do2();
}
```

[MISRA, Rule 59][JSF AV, Rule 59][CERT, EXP19-C]

3.3 Rekursion

Einige Standards schreiben vor das Rekursion nicht benutzt werden soll, weil man den Speicherverbrauch und die Laufzeit nicht genau bestimmen kann.[MISRA, Rule 70] Andere sagen das wenn man diese Werte kennt, sie dennoch benutzen kann.[JSF AV, Rule 119] Abgesehen davon können die meisten Probleme auch iterativ gelöst werden.

3.4 Heap Operationen

Zu den Heap Operationen zählen malloc, realloc, free, new. Nachteile sind unter anderem das der Speicherverbrauch schwer zu bestimmen ist, vor allem wenn sie nach der Initialisierung, also in der Hauptschleife des Programmes benutzt werden. Auch kann es durch diese Operationen zur Fragmentierung des Heaps kommen, wodurch je nach Implementierung der Operationen unbekannte Laufzeiten entstehen. Auch hier herrscht Uneinigkeit zwischen den Standards. Bei einigen ist das benutzen dieser Operationen komplett verboten.[MISRA, Rule 118] Bei anderen ist es bei der Initialisierung erlaubt.[JSF AV, Rule 206] Bei dem nicht auf Embedded Software zugeschnittenem Standard ist die Verwendung von diesen Operationen in einem eigenen Kapitel geregelt, aber grundsätzlich erstmal erlaubt.[CERT, MEMxx]

3.5 Funktionen

3.5.1 Fehlererkennung

Es sollen nur Funktionen verwendet werden, deren Rückgabewerte eindeutig erkennen lassen das zum Beispiel ein Fehler aufgetreten ist. In allen Standards sind viele Funktionen explizit aufgelistet die nicht benutzt werden dürfen die genau dieses Verhalten zeigen.

Listing 5: Falsch

```
int32_t a;
a = atoi( "0" );
if(a == 0){ //ist a wirklich 0 oder ist ein Fehler aufgetreten?
    printf("error");
}
```

Listing 6: Richtig

```
int32_t a;
a = (int)strtol("0", (char **)NULL, 10);
//gibt auch 0 bei einigen Fehlern zurueck
//aber es kann ueberprueft werden ob wirklich ein Fehler aufgetreten ist
```

[MISRA][JSF AV][CERT]

3.5.2 Reentrancy

Reentrancy ist in Konzept für Funktionen das ursprünglich dazu gedacht war Code auf Systemen mit wenig Speicher zu teilen. Genauer gesagt geht es darum das es keine Daten gibt die bei mehreren gleichzeitigen Aufrufen geteilt werden. Der Vorteil ist, dass bei einer Unterbrechung der Funktion und erneutem Aufruf z.B. durch einen Interrupt keine Fehler entstehen. Deswegen müssen rekursive Funktionen, wenn sie denn verwendet werden, reentrant sein. Die folgenden drei Punkte müssen dabei von Funktionen die reentrant sind erfüllt werden.

- alle globalen Variablen müssen atomar benutzt werden
- die Hardware muss atomar benutzt werden
- es werden nur Funktionen aufgerufen die ebenfalls Reentrant sind

[Labrosse, S. 655f]

3.6 Switch

Bei einem switch sollten immer alle Fälle als case existieren und zusätzlich auch ein Default Eintrag. Falls nicht für alle Fälle ein case existiert sollte dies dokumentiert werden. Außerdem muss jedes case mit einem break abgeschlossen werden. Diese Regeln

stellen sicher das jeder mögliche Wert abgedeckt ist und das nicht Code ausgeführt wird der nicht zum jeweiligen Fall gehört. Einige Standards schreiben auch vor das ein switch nur über einen enum ausgeführt werden darf, was die Anzahl der Fälle im Gegensatz zu z.B. Integern stark verringert. [MISRA, Rule 61,62][JSF AV, Rule 194,196][CERT, MSC01-C,MSC17-C,MSC20-C]

3.7 Optimierungen

Code sollte, wenn überhaupt erst am Ende und nach ausgiebigen testen optimiert werden. Denn meistens optimiert der Compiler den Code schon sehr gut. Oft sogar besser als ein Mensch das könnte. Ein weiterer Grund die Optimierungen erstmal sein zu lassen sind lesbarer und vor allem fehlerfreier Code. Denn was nützt Code der schnell ist, aber unleserlich oder sogar fehlerhaft.

Eine weitere Sache die gerne in der Embedded Programmierung benutzt wird ist inline Assembler, weil dieser angeblich schneller ist. Das Problem dabei ist, dass der Compiler den Code um den Assambler herum nicht oder nur sehr schlecht optimieren kann. Deswegen sollte Assembler, wenn überhaupt, in extra Funktionen gepackt werden, so dass der Compiler um den Aufruf herum optimieren kann. [Labrosse, S. 691f][JSF AV, Rule 216]

4 Beispiele für vorgegebenen Code

Als Richtlinien können auch Codevorlagen für bestimmte Probleme existieren. Im [CERT] zum Beispiel werden viele Lösungen für häufige Probleme gegeben.

4.1 Interrupts

Interrupts sind schön und gut, aber manchmal möchte man sie gerne abschalten, doch wie macht man dies richtig?

Listing 7: Fehler

```
int a = 0;
void Foo() {
  disableInterrupts();
  a += 5;
  enableInterrupts();
}
```

Das Problem ist klar, wenn Interrupts vor dem Aufruf von Foo aus waren, so sind sie nachher an, was irgendwann zu Fehlern führen wird.

Listing 8: Besser

```
int a = 0;
void Foo() {
  saveInterruptState();
```

```
    disableInterrupts();
    a += 5;
    loadInterruptState();
}
```

Hier wird das Problem umgangen in dem der Zustand ob Interrupts an oder aus waren gespeichert und am Ende wieder geladen wird. Einen Nachteil gibt es dennoch bei Systemen mit reellen oder simuliertem Multithreading: Foo muss atomar sein, sonst wäre es möglich, dass die Reihenfolge in der der Zustand gespeichert und geladen wird sich vertauscht und dadurch die gleichen Probleme entstehen wie bei der fehlerhaften Variante. [Labrosse, S. 656-659]

4.2 Kritischer Bereich

Als kritischer Bereiche werden Abschnitte im Programm bezeichnet die während der Ausführung nicht von mehr als einem Thread betreten werden dürfen. Die Absicherung solcher Abschnitte wird oft fehlerhaft vorgenommen, dabei gibt es eine einfache Lösung die fast immer verfügbar ist: test-and-set.

Listing 9: Fehler

```
while(wait);
wait = TRUE;
...
wait=FALSE;
```

Direkt nachdem wait auf false gesetzt wird besteht die Möglichkeit für mehr als einen Thread bzw. Interrupt den Bereich zu betreten.

Listing 10: Richtig

```
while(TestAndSet(wait));
...
wait=FALSE;
```

Da TestAndSet atomar ist kann der kritische Bereich nur einmal betreten werden. Falls der Prozessor keine TestAndSet Anweisung zur Verfügung stellt, kann diese auch durch mehrere Assemblerbefehle nachgebaut werden.

Listing 11: Assembler für TestAndSet

```
loop:
    mov     a1, 0           ; 0 = wird benutzt
    xchg    a1, variable
    cmp     a1, 0
    je      loop
```

Falls der Vergleich ungleich ergibt, also in variable etwas anderes als 0 stand, wird TestAndSet verlassen und der kritische Bereich betreten. Einzige Voraussetzung ist das

es einen atomaren Befehlt gibt der den Inhalt eines Registers mit dem einer Variable tauschen kann.
[Labrosse, S. 656f, 660f]

5 Eigene Erfahrungen

Durch die Module SWTP(Softwaretechnikpraktikum) und MPCP(Mikroprozessor- und Controllerpraktikum) im letzten Semester konnte ich einige Erfahrung bei der Embedded Programmierung gewinnen. Deswegen werden in diesem Kapitel nicht nur Richtlinien der Standards beschrieben, sondern auch die Erfahrungen, die ich in diesen beiden Modulen gemacht habe.

5.1 Modularisierung

Ein ganz wichtiger Punkt, der natürlich bei jeder Art von Projekt beachtet werden sollte ist die Modularisierung. Denn wenn die Anwendungslogik einmal mit der Hardware verstrickt ist, wird man dies nur noch schwer lösen können. Deswegen ist es wichtig diese beiden Bereiche klar voneinander zu trennen. Bei der Embedded Programmierung kann man zum Beispiel auf CMSIS zurückgreifen, welches die Hardware um eine weitere Stufe abstrahiert. Die Vorteile der Modularisierung sind die Möglichkeit die Logik auf einem anderen System zu testen und es einfacher auf ein anderes System zu portieren. Denn so muss nur die Hardwareschicht ausgetauscht werden, und nicht die komplette Programmlogik. [JSF AV, S. 10f] Beim SWTP, wo wir als Gruppe eine Ampelsteuerung entwickeln mussten haben sich klar die Vorteile der Modularisierung gezeigt. Die Logik der Ampel und die Zugriffe auf die Hardware waren in unterschiedlichen Modulen untergebracht. So konnte die Logik auf Windows/Linux Rechnern mit Hilfe eines Testframeworks überprüft werden. Auch als wir gewisse Einschränkungen der Hardware feststellten musste nur die Hardwareschicht ausgetauscht werden, die Logik blieb davon unberührt.

5.2 Was kann der Compiler bzw. die Hardware?

Ein Problem was sowohl bei SWTP als auch MPCP auftrat war, dass der Code ohne Fehler compiliert wurde, bei der Ausführung aber Fehler auftraten. Dies äußerte sich z.B. bei malloc/new so, dass Code für diese Funktionen generiert wurde, dieser aber bei jeder Ausführung die Speicheradresse 0 zurücklieferten, obwohl Speicher frei war. Bei SWTP traten weitere Probleme dieser Art auf, vor allem bei C++ Features wie virtual oder namespaces. Es gab aber auch noch andere Probleme wie Fehlerhafte Dokumentation der Hardware. Angeblich standen 255 Interrupts zur Verfügung, mehr als einer funktionierte aber nicht gleichzeitig und das trotz 16 verschiedenen Prioritätsstufen. Was man davon lernen kann ist folgendes: Wenn man mit Hardware arbeitet die man vorher noch nie benutzt hat, sollte man Testen, ob diese alles unterstützt was gebraucht wird, denn nicht alles was in der Dokumentation/Referenz steht oder compiliert ist richtig und funktioniert auch.

5.3 Kommentare

Kommentare sind wichtig. Wie viel und was genau kommentiert wird hängt davon welche Regeln man dafür benutzt. Dies unterscheidet sich teilweise doch sehr deswegen keine genauen Beispiele dazu. Aber bei Embedded Programmierung sollte man auf einige Dinge achten. Wenn man auf die Hardware zugreift, hat man oft genug Konstanten, manchmal aber auch nicht. In beiden Fällen ist es sinnvoll zu kommentieren was getan wird, damit der nächste der den Code sieht nicht erst in der Referenz suchen muss. Am besten gibt man auch noch einen Verweis auf die Referenz mit an, dann passiert so etwas auch gar nicht erst. [MISRA, Rule 9,10][JSF AV, Rule 126-134][CERT, MSC04-C]

Literatur

[Labrosse] Jean Labrosse u.a. Embedded Software: Know It All, 1997

[CERT] https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards (15.05.2011)

[Misra Website] http://www.misra-c.com (15.05.2011)

[JSF AV] http://www2.research.att.com/~bs/JSF-AV-rules.pdf (15.05.2011)

[Stroustrup] http://www2.research.att.com/~bs/ (15.05.2011)

[MISRA] http://computing.unn.ac.uk/staff/cgam1/teaching/0703/misra\%20rules.pdf (15.05.2011)

[EC++] http://de.wikipedia.org/wiki/Embedded_C\%2B\%2B (15.05.2011)