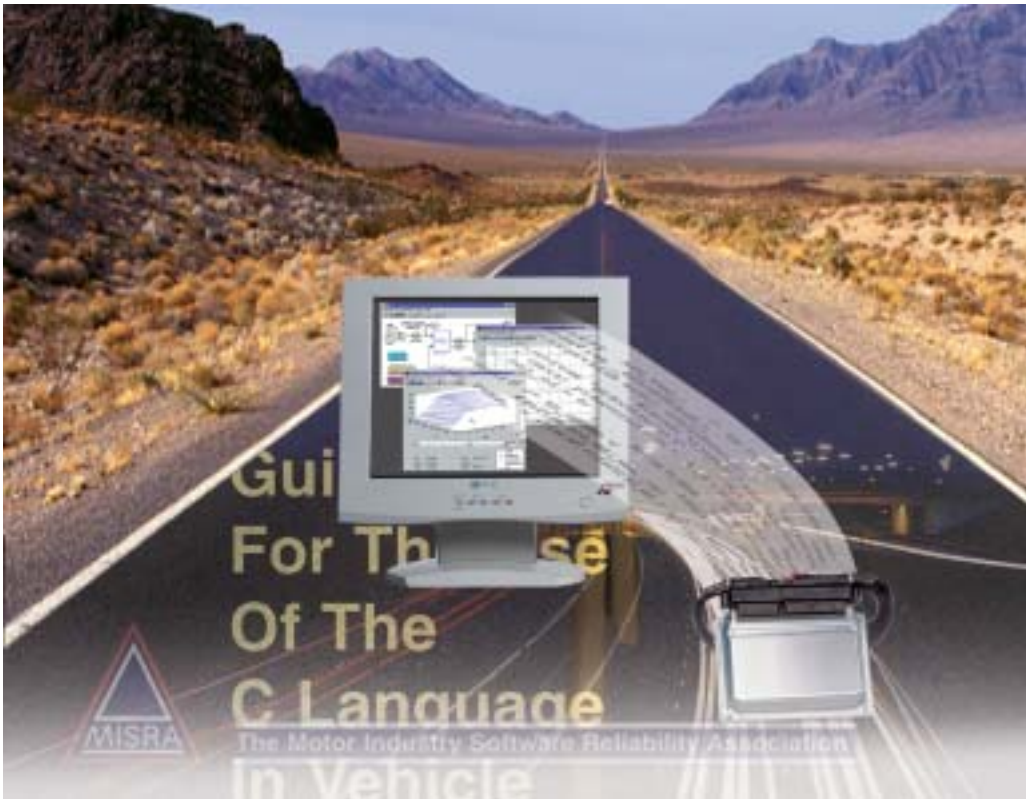

MISRA C und seine Anwendbarkeit auf Seriencodegeneratoren

MISRA C ist ein Codierungsstandard, welcher in der Automobil-Industrie immer häufiger angewendet wird. Mit dem Aufkommen von Seriencodegeneratoren stellt sich die Frage, ob MISRA C auch auf diese anwendbar ist. Der Standard wurde für Software-Entwickler geschrieben, um übliche Programmierfehler vermeiden zu helfen. Diese potentiellen Fehler sind nicht notwendigerweise die Gleichen, welche bei maschinengeneriertem Code auftauchen können.

Der Artikel beschreibt das Konfliktfeld zwischen dem Wunsch nach vollständiger Konformität zu einer Norm, der Sinnhaltigkeit seiner Regeln und die Kosten für die Einhaltung seiner Regeln. Letzteres wird durch aktuelle Benchmarks untermalt, welche mit dem Seriencodegenerator TargetLink von dSPACE erzeugt wurden. Die Beispiele mit TargetLink zeigen im Besonderen, dass die meisten Regeln eingehalten werden können, - aber auch dass einige Regelausnahmen sehr ratsam sind, um den generierten Code effizient zu erhalten und ohne dass dabei Kompromisse bei der Code-Sicherheit eingegangen werden müssen.

Einleitung

Seit 1998 gibt es von der britischen „Motor Industry Software Reliability Association“ [1], kurz MISRA, einen allgemein zugänglichen Standard für die Verwendung von C in Steuergeräteprojekten bis SIL3. Er wird unter dem Titel „Guidelines for the use of the C language in vehicle based software“ [2] geführt, ist aber allgemein unter dem Schlagwort MISRA C bekannt. Firmen aus der Automobilindustrie übernehmen den Standard zunehmend als Ersatz oder Ergänzung ihrer internen Standards. Er wird in Kundenprojekten von den Auftraggebern vorgeschrieben, so dass Zulieferer zur Übernahme des Standards praktisch gezwungen werden. Automatische Code-Inspektionstools existieren reichhaltig auf dem Markt und können mit einem gewissen Maß an Zuverlässigkeit Regelverletzungen in einer C-Datei erkennen. Bekanntere Tools mit einem MISRA-C-Modul sind unter anderem QA C von QA-Systems und Testbed von LDRA [3].

Die Autoren des MISRA-C-Standards haben das verfügbare Wissen aus Literatur [4], Firmenstandards und der Praxis zusammengetragen. Ziel des Standards ist es, Regeln zur Vermeidung von üblichen Software-Fehlern aufzustellen. Die Fehler stammen aus den folgenden Kategorien:

- Übliche Programmierfehler
- Nichtbeachtung von Lücken in der Sprachendefinition
- Missverständnis der Programmiersprache
- Missverständnis des Compilers
- Compilerfehler und Laufzeitfehler

Die ersten vier der fünf Fehlerklassen beziehen sich auf menschliches Versagen. Damit ist die Zielgruppe, auf welche sich der Standard bezieht, recht klar definiert: der Mensch als Software-Entwickler.

Probleme mit Normen treten immer dann auf, wenn sie für Zwecke verwendet werden, für die sie nie vorgesehen wurden. Das ist hier der Fall. MISRA C wurde für hand-geschriebenen C Code entwickelt und wird nun auch für maschinen-generierten C Code angewendet. Code-Generatoren machen mit einem überzeugenden Grad an Zuverlässigkeit bestimmte Fehler nicht, die für Menschen typisch sind. Der Nutzen bestimmter MISRA-C-Regeln ist dann nicht mehr gegeben. Übrig bleibt dann bestenfalls eine wirkungslose Regel. Andernfalls haben die Regeln einen negativen Effekt auf die Code-Effizienz oder lassen sich durch eine Maschine nicht erfüllen. Im folgenden Abschnitt werden hierzu einige Beispiele erörtert.

Hardwarenaher Code

Wenn einige MISRA-C-Regeln Probleme bei der Code-Generierung bereiten, dann resultiert das hauptsächlich in Verlust an Code-Effizienz, ohne dass sich der erklärte Nutzen der Regel einstellt. Verlust der Code-Effizienz bedeutet einen Mehrbedarf an ROM, RAM, Stack oder Laufzeit.

Bereits Regel 1 des MISRA-C-Standards ist für die Code-Generierung problematisch. Sie ist eine Pflichtregel und verbietet die Verwendung von compilerspezifischen Spracherweiterungen. Es wird keine Erklärung für die Regel gegeben, sondern nur auf einen anderen Standard verwiesen, der die Verwendung einer „standardisierten, strukturierten Sprache“ vorschreibt.

Hintergrund der Regel ist die Portabilität des C-Codes. Dieses Problem ist bei Code-Generatoren nicht mehr relevant. Der Input eines Code-Generators sind Blockdiagramme, die in der Regel keine Hardware-Abhängigkeiten kennen. Der generierte C-Code ist nur noch ein Zwischenformat, welches jederzeit neu generiert werden kann. Serien-code-Generatoren – wie TargetLink [5] – besitzen spezifische Target-Optimierungsmodule, die hocheffizienten Code unter Ausnutzung von compilerspezifischen Spracherweiterungen liefern können. Wenn sich der Prozessor oder Compiler für ein Entwicklungsprojekt ändert, muss lediglich das Target-Optimierungsmodul des Code-Generators ausgetauscht werden. Falls es das nicht gibt, kann immer noch portabler ISO-C-Code erzeugt werden, welcher dann wieder der MISRA-C Regel 1 entspricht. Diese Flexibilität der Code-Generatoren und die Hardware-Unabhängigkeit von Blockdiagrammen machen die Regel 1 überflüssig.

Regel 3 ist verwandt mit der Regel 1. Letztere verbietet Assembler Code, welcher eine compilerspezifische Spracherweiterung ist. Falls dennoch Assembler verwendet werden muss, gibt Regel 3 Einschränkungen vor. Assembler Code und C-Code sollen nicht vermischt werden und Assembler Code soll mit einem C-Funktionsaufruf gekapselt werden. Eine Begründung wird nicht gegeben.

MISRA-C Regel 1 eingehalten:
Die Sättigung erfolgt per Plausibilitätscheck im ISO-C-Code.

Code:

```
e = REF - POS;
if ((REF >= 0) && (POS < 0) && (e < 0)) {
   e = 32767;
} else {
   if ((REF < 0) && (POS >= 0) && (e >= 0)) {
      e = -32768;
   }
}
```

Laufzeit:	1,1 µs	Prozessor:	Infineon TriCore, 40 MHz
Code-Größe:	48 Bytes	Compiler:	Tasking v. 1.3r1

MISRA-C Regel 1 nicht eingehalten:
Der Castingoperator „`_sat`“ wird verwendet, um Arithmetikoperationen per Hardware zu sättigen.

Code:

```
e = (_sat Int16)((_sat Int16)REF-(_sat Int16)POS);
```

Laufzeit:	0,3 µs	Prozessor:	Infineon TriCore, 40 MHz
Code-Größe:	12 Bytes	Compiler:	Tasking v. 1.3r1

Beispiel 1: *Sättigung beim Infineon TriCore. Mit einer compilerspezifischen Spracherweiterung lassen sich 72% Laufzeit und 75% Code-Größe sparen.*

MISRA-C Regeln 1 und 3 eingehalten:
Eine 64-Bit Multiplikation wird mittels einer ISO-C-Funktion berechnet.

Laufzeit:	32,6 µs	Prozessor:	Motorola M68332, 16 MHz
Code-Größe:	32 + 142 (library) Bytes	Compiler:	Microtec v. 4.5r

MISRA-C Regeln 1 und 3 nicht eingehalten:
Eine 64-Bit Multiplikation wird mittels einem Assemblermakro berechnet.

Laufzeit:	5,5 µs	Prozessor:	Motorola M68332, 16 MHz
Code-Größe:	26 Bytes	Compiler:	Microtec v. 4.5r

Beispiel 2: *Inline-Assemblermakros beim Motorola M68332. Durch die Verwendung der Assemblersprache in einem Makro lassen sich 83% Laufzeit und 85% Code-Größe sparen.*

Damit wird die Verwendung von Makros ausgeschlossen. Funktionsaufrufe verbrauchen zusätzliche Rechenzeit, was bei Makros nicht der Fall ist. Durch die Regel 3 wird der Laufzeitvorteil, den man durch die Verwendung von Assembler erzielen kann, wieder zunichte gemacht. Dieses gilt insbesondere für häufig aufgerufene Funktionen, wie es beispielsweise Arithmetikoperationen mit doppelter Wortbreite sein können.

Regel 28 verbietet die Verwendung der Speicherklasse ‚register‘. Die Begründung dazu lautet, dass dieses nur eine Empfehlung an den Compiler sei und gute Compiler die richtige Registerbelegung selber finden ‚sollten‘. Letzteres ist leider nicht immer der Fall. Während der Entwicklung von TargetLink-Optimierungsmodulen sind verschiedene Fälle gefunden worden, wo die Verwendung von ‚register‘ das Anlegen von Variablen im RAM verhindert hat. Das spart Ressourcen in allen drei Disziplinen: Eine RAM-Variable wird weniger erzeugt, es müssen keine Adressen für die Variable im Opcode gespeichert werden und es gibt auch keinen RAM- Zugriff während der Laufzeit.

Nebenbei sei bemerkt, dass man mit der Deklaration einer Registervariable den späteren Zugriff mittels Pointer verhindern kann. Geschieht dieses im Code,

MISRA-C Regel 28 eingehalten:

Die Arithmetikoperation wird mittels einem Assemblermakro auf dem Stack ausgeführt.

Code:

```
#define AC__I16SUBI16I16_SAT(s1, s2, csatval, r)    \
{                                                   \
   Int16 tmp_s1 = s1;                               \
   Int16 tmp_s2 = s2;                               \
   r = asm(Int16,   " move.w `tmp_s1`,D0",          \
                    " sub.w  `tmp_s2`,D0",          \
                    " bvc *+16",                    \
                    " tst.w D0",                    \
                    " blt *+8",                     \
                    " move.w #-"#csatval"-1,D0",    \
                    " bra *+6",                     \
                    " move.w #"#csatval",D0");      \
}
```

Laufzeit:	3,1 µs	Prozessor:	Motorola M68332, 16 MHz
Code-Größe:	38 Bytes	Compiler:	Microtec v. 4.5r

MISRA-C Regel 28 nicht eingehalten:

Die Arithmetikoperation wird mittels einem Assemblermakro in Prozessorregistern ausgeführt.

Code:

```
#define AC__I16SUBI16I16_SAT(s1, s2, csatval, r)    \
{                                                   \
   register Int16 tmp_s1 = s1;                      \
   register Int16 tmp_s2 = s2;                      \
   r = asm(Int16,   " move.w `tmp_s1`,D0",          \
                    " sub.w  `tmp_s2`,D0",          \
                    " bvc *+16",                    \
                    " tst.w D0",                    \
                    " blt *+8",                     \
                    " move.w #-"#csatval"-1,D0",    \
                    " bra *+6",                     \
                    " move.w #"#csatval",D0");      \
}
```

Laufzeit:	2,1 µs	Prozessor:	Motorola M68332, 16 MHz
Code-Größe:	30 Bytes	Compiler:	Microtec v. 4.5r

Beispiel 3: *Sättigung beim Motorola M68332. Durch das ‚register'-Attribut bei der Deklaration von Zwischenvariablen lassen sich 32% Laufzeit und 21% Code-Größe sparen.*

MISRA-C Regel 101 eingehalten:

Index-Suchfunktion ohne Pointerarithmetik.

Laufzeit:	26,1 µs	Prozessor:	Motorola HC12, 8 MHz
Code-Größe:	105 Bytes	Compiler:	Cosmic v. 4.2u

MISRA-C Regel 101 nicht eingehalten:

Index-Suchfunktion mit Pointerarithmetik.

Laufzeit:	22,9 µs	Prozessor:	Motorola HC12, 8 MHz
Code-Größe:	94 Bytes	Compiler:	Cosmic v. 4.2u

Beispiel 4: *Verwendung von Pointerarithmetik für die Index-Suchfunktion eines Kennfeldes beim Motorola HC12. Mit einem Pointer für die Adressierung des Stützstellenvektors lassen sich 12% Laufzeit und 10% Code-Größe sparen.*

würde es eine Fehlermeldung während der Kompilierung geben. Das wäre in einigen Fällen eine berechtigte Sicherheitsmaßnahme, welche die Regel 28 aber nicht erlaubt.

Typische Implementierungen von Arithmetik-Operationen

Regel 37 verbietet die Anwendung von Bit-Operationen auf ‚signed integer'-Datentypen. Der Grund ist offensichtlich. Nur zu leicht können solche Operatoren das Vorzeichenbit unerwünscht verändern. Der Fehler wird leicht gemacht und ist schwer zu finden. Um Regel 37 einzuhalten, muss der Software-Entwickler statt eines Bitshift-Right nun eine Division vornehmen. Damit kann ein unerwartet gesetztes Vorzeichenbit nicht mehr in den numerischen Teil der Variable gelangen und somit das Ergebnis verfälschen. Andererseits dauern Divisionen auf üblichen Prozessoren erheblich länger als Bitshifts.

Heutige Prozessoren besitzen Arithmetik Bitshift Befehle, um dieses Problem zu begegnen. Eine Ergebnisverfälschung ist dadurch nicht mehr möglich. Muss eine Regel eingehalten werden, obwohl man mit Sicherheit weiß, dass der Fehler „per Design" nicht mehr auftreten kann? Divisionen kosten bei üblichen Prozessoren deutlich mehr Rechenzeit als Bitshifts. Eine Befolgung der Regel 37 hat jetzt nur noch die Folge, dass sich die Laufzeit des generierten Codes erhöht.

Regel 101 verbietet Pointerarithmetik. Die Problematik ist sicher jedem SoftwareExperten (wohl auch aus persönlicher Erfahrung) bekannt. Trotzdem ist die Verwendung von Pointerarithmetik in Suchalgorithmen und Interpolationsroutinen für Steuergerätekennfelder absolut üblich. Bei einigen DSP-Routinen können Laufzeitrestriktionen ohne Pointerarithmetik nicht mehr eingehalten werden.

Anstelle Indexvariablen zu inkrementieren und dann Array-Felder damit zu adressieren, können, mittels Inkrement- oder Dekrementbefehlen auf Pointer angewendet, die gleichen Elemente eines Arrays deutlich effizienter adressiert werden. Manche Prozessoren bieten für diese Operationen

spezielle Assembler-Befehle an, die vom Compiler nur dann verwendet werden, wenn der C-Code spezielle C-Codemuster mit Pointerarithmetik enthält.
Von einem Code-Generator kann erwartet werden, dass er keine Fehler in der Pointerarithmetik macht. Wegen der Bedeutung von Kennfeldern in Steuergeräten gehört die Regel 101 mit zu den folgenschwersten Regeln für die Code-Effizienz, wenn sie von einem Code-Generator befolgt werden müsste.

Mehrfachverwendung und Typenkonvertierung von Variablen

Regel 109 verbietet die Mehrfachverwendung von Variablen. Dieses ist jedoch bei temporären, lokalen Variablen sinnvoll, um den Stackverbrauch zu minimieren. Mittels einer Lebensdaueranalyse stellt der Code-Generator sicher, dass sie in einem Code-Abschnitt nicht mehr gebraucht wird, bevor ihr eine neue Bedeutung zugewiesen wird. Code-Generatoren machen hier keine Fehler. Auch Änderungen im Blockdiagramm stellen keine späteren Fallen dar, weil die Lebensdaueranalyse bei jedem Lauf des Code-Generators wiederholt wird.
Die Kombination der Regeln 43 und 44 ist für generische ISO-C-Code-Generierung nicht erfüllbar. Regel 43 schreibt vor, dass es keine impliziten Typenkonvertierungen im Code geben darf und Regel 44 verbietet überflüssige, explizite Typenkonvertierungen. Beide Regeln sind nur zu erfüllen, wenn die Bitbreite von ‚char', ‚short', ‚int' und ‚long' bzw. ‚float' und ‚double' bekannt sind, was wiederum die Kenntnis des verwendeten Compilers und Prozessors voraussetzt. Im generischen ISO-C-Modus von TargetLink ist dieses nicht bekannt und der Code-Generator muss portablen Code für alle möglichen Bitbreiten generieren. Somit werden tendenziell zu viele Casting-Operatoren im Code erzeugt, die bei spezifischen Compiler/Prozessor-Kombinationen nicht nötig wären.

Übertragen von MISRA C Regeln auf die Modellebene

Regel 50 verbietet den exakten Vergleich von Fließkommavariablen auf Gleichheit oder Ungleichheit. Regel 70 verbietet die Verwendung von rekursiven Funktionsaufrufen. Beide Regeln lassen sich auf Blockdiagrammebene transferieren. So kann Regel 50 immer dann erfüllt werden, wenn der Anwender im Blockdiagramm den ‚Relational Operator' Block nicht mit Gleichheit oder Ungleichheit für Fließkommasignale verwendet. Regel 70 wird immer dann eingehalten, wenn der Anwender auf ‚Undirected Event Broadcasts' in einem Statechart verzichtet.
Weil nicht alle Anwender von TargetLink nach MISRA C entwickeln müssen, hat dSPACE die Code-Generierung für diese Fälle zugelassen. Anwender, die MISRA-C-Code abzuliefern haben, müssen die beschriebenen Einschränkungen auf Blockdiagrammebene berücksichtigen.
Weitere Regeln des MISRA-C-Standards lassen sich einfach durch die richtige Wahl von Optionen in TargetLink erfüllen. Beispielsweise ist die Begrenzung der Zeichenlänge für Symbole auf 31 signifikante Stellen (Regel 11) im TargetLink- Maindialog einstellbar, ebenso wie die Generierung von reinem ISO-C-Code (Regel 1). Falls #define/#undefine-Direktiven nicht in einem Codeblock auftauchen sollen (Regel 91), kann in der TargetLink-Konfiguration die Variablenklasse LOCAL_MACRO entfernt werden.

MISRA C-Konformität von TargetLink

Die oben angeführten Probleme mit dem MISRA-C-Standard und der Code-Generierung sind zahlenmäßig eher gering im Vergleich zu der Gesamtzahl der Regeln. Der Standard beinhaltet insgesamt 127 Regeln. TargetLink von dSPACE erfüllt davon 92 Regeln. Weitere 7 Regeln werden immer befolgt, wenn TargetLink richtig konfiguriert wird. 3 Regeln können eingehalten werden, wenn sich der Anwender auf Blockdiagrammebene einschränkt. 5 Regeln werden von TargetLink größtenteils eingehalten. Lediglich 20 Regeln werden komplett nicht eingehalten. Davon sind 12 geforderte (‚required') Regeln und 8 sind empfohlene (‚advisory') Regeln. dSPACE kann einige von ihnen in zukünftigen Versionen implementieren. Die Mehrzahl hat aber die oben beschriebenen Nachteile, welche eine Implementierung nicht ratsam erscheinen lässt.

Gesamtzahl der MISRA-C-Regeln: 127

davon:

... voll erfüllt	92
... erfüllt bei richtiger Tool-Konfiguration	7
... erfüllt bei richtigem Modellierungsstil	3
... teilweise erfüllt	5
... nicht erfüllt	20

Tabelle 1: *Erfüllungsmatrix von MISRA-C-Regeln in TargetLink*

MISRA erlaubt ausdrücklich Abweichungen vom Standard, wenn diese technisch begründet und dokumentiert sind. Für TargetLink gibt dSPACE ein solches Dokument heraus.

Ausblick

Das Dokumentieren von Abweichungen von einem Standard ist eine praktikable Lösung, aber es ist keine befriedigende Lösung. Das gilt insbesondere für solche Fälle, wo die Abweichungen nur deshalb zu Stande kommen, weil der Standard noch nicht den neusten Stand der Technologie berücksichtigt. Eine Aktualisierung des Standards ist dann dringend notwendig.
Eine zufriedenstellende Lösung im MISRA C Standard wäre eine Unterscheidung der Regeln nach Gültigkeit für menschliche Programmierer und Code-Generatoren. Letzteres wird wahrscheinlich eine Teilmenge des gesamten Regelwerkes sein. Mit einer solchen Änderung könnte der MISRA C Standard problemlos solchen Anwendern dienen, welche sowohl moderne Seriencodegeneratoren als auch handgeschriebenen Code einsetzen.

Literatur

[1] www.misra.co.uk

[2] MISRA Guidelines for the Use of the C Language in Vehicle Based Software, April 1998

[3] SP & JJ:
"A Comparison of MISRA C Testing Tools", PI Technology, 2001

[4] Hatton, L.:
"Safer C : Developing Software for High-Integrity and Safety-Critical Systems"
McGraw-Hill, 1995

[5] Hanselmann, H./ Kiffmeier, U./ Köster, L./ Meyer, M.:
"Automatic Generation of Production Quality Code for ECUs"
SAE Technical Paper 99P-12, 1999

Der Autor

Dipl.-Ing. Thomas Thomsen hat an der Universität Karlsruhe Elektrotechnik studiert und 1996 im Fach Industrielle Informationstechnik abgeschlossen. Anschließend arbeitete er als Entwicklungsingenieur und Projektleiter für die ABS/TCS Steuergeräte-Entwicklung bei der Continental-Teves in Deutschland und in den USA. 1999 wechselte Thomas Thomsen zur dSPACE GmbH in Paderborn und ist seitdem als Produktmanager für den Bereich Seriencodegenerierung verantwortlich.

Kontakt

Thomas Thomsen
Produktmanager TargetLink
dSPACE GmbH
Technologiepark 25
33100 Paderborn
Deutschland

tthomsen@dspace.de
http://www.dspace.de