Vol.2012-SE-176 No.4
Vol.2012-EMB-25 No.4
2012/5/21

組込みソフトウェアの MISRA-C 対応の方法について

中野俊和[†]　　赤星　博輝[†]

近年、組み込みソフトウェアでは、IEC 61508 や ISO 26262（特にソフトウェア開発の Part6）への対応による品質確保の取り組みが行われている。弊社でも品質向上に対する取り組みが重要であると判断し、ソフトウェアの静的チェックとして MISRA-C を適用することにした。しかしながら、MISRA-C には曖昧なルールやマイコン/コンパイラに依存する部分があり、適用に際して実際にどのように運用するかを決定する必要がある。本書では、主に移植性に影響する演算子の問題と、マシン依存のライブラリを使用した問題の解決策について、MISRA-C の運用の仕方を検討する。

A study of MISRA-C coding style for embedded software systems

TOSHIKAZU NAKANO[†]　HIROKI AKABOSHI[†]

Recently, embedded software systems have several challenges to improve its quality and to comply IEC 61508 and ISO 26262(especially Part6 for software development). Coding guideline is important to accomplish the challenges and then we adopt MISRA-C for the coding guideline. Some MISRA-C rules are ambiguous and are depend on microprocessors/compilers, then all rules should be examined and coding style should be defined to achieve MISRA-C compliance for our projects. This report shows the examination of the rules, especially C language operators problems for portability, and the solution to the problems using machine-depending library.

1. はじめに

近年、組込みソフトウェアでは、品質に関する取り組みが再度見直されている。その理由としては、現実のソフトウェアに対する品質確保が年々難しくなっており、現実での問題発生や IEC 61508 や ISO 26262 といった機能安全規格への対応などが挙げられる。

弊社でもソフトウェアの品質向上に対する取り組みが必要であると判断し、ソフトウェアの静的チェックとして MISRA-C を適用することに決定した。しかしながら MISRA-C には曖昧なルールやマイコン/コンパイラに依存する部分があり、適用に際して実際にどのように運用するかを決定する必要がある。今回 MISRA-C についてルールを精査し、課題を整理した上で、運用の仕方について検討を行った。

2. 社内課題

社内のソフトウェア技術者は中途入社の比率が多く経験してきた業務領域などが異なりこれまで経験してきたソフトウェアの作成方法/コーディングルール/テスト手法などが異なることが課題としてあった。今回こうしたバラバラの状態を解消するため、事業部としては MISRA-C : 2004（以後 MISRA-C）を C 言語のガイドラインとして採用し、今後必要とされるプロジェクトに適用することにした。

MISRA-C[1][2]では 21 のカテゴリに 141 のルールがあり、その中で 121 は必要ルール、残り 20 は推奨ルールとなっている。必須ルールは強制的に守ることが求められており、ルールに従えない場合には正式な逸脱の手続きが必要となっているのに対して、推奨ルールは通常従う方が良いとされているがルールに従えなくても逸脱の手順は不要とされている。ルールによっては読み手によって理解が異なる点や対応方法に差が出るものもあった。

こうした点からも単純に MISRA-C 対応という方針を決めただけでは、必要や推奨および逸脱などの判断でばらつきが出てしまうと考え、MISRA-C のルールについて内容を精査し、対応などについて検討を実施した。

3. MISRA-C ルール精査

MISRA-C のルールを精査する際に、設計時、メンテナンス時、マイコン/コンパイラ変更時という視点から、141 のルールに対して以下の 3 つの観点から評価を行い、対応について検討した。1 つのルールが以下の複数の視点にあてはまることがある。

- 間違い易い記述
- 設計意図が分かり難い記述
- マイコン/コンパイラ依存の記述

[†](株)ガイア・システム・ソリューション，オートモーティブ事業部
GAIA System Solution Inc.

Vol.2012-SE-176 No.4
Vol.2012-EMB-25 No.4
2012/5/21

3.1 間違い易い記述

間違い易い記述としては、設計者がミスをしやすい点について検討を実施した。

例えば、ルール 5.2 は“外部スコープの識別子が隠蔽されることになるため、内部スコープの識別子には、外部スコープの識別子と同じ名前を用いてはならない”。記述例 1 にあるようにグローバル変数とオート変数で同じ名前を使うことで、ミスし易い状況が発生する。

（記述例 1）

```
int     value;
void    foo (    ) {
     int value;
     ・・・
}
```

ルール 14.9 は“if 構造の後には複合文を続けなければならない。else キーワードの後には複合文または他の if 文を続けなければならない”。記述例 2 では、この記述だけ見れば C 言語としては問題なく見えるが、機能追加などを行ったときに括弧｛｝の追加を忘れたりする危険性がある。

（記述例 2）

```
if    ( a == 0x0 )
   x = 0x00;
```

この視点では、細かい点での反論などはあったものの、基本的には逸脱しない方向で対応することに決定した。

3.2 設計意図が分かり難い記述

設計意図が分かり難い記述は、記述からは設計者がミスしたのか、あるいは、意図的にそのような記述をしたのか分かり難い記述についてのルールについて検討を実施した。

例えば、ルール 13.1 は“ブール値が生成される式の中で代入演算子を用いてはならない”であり、記述例 3 のような記述を禁止している。これは、記述だけでは t_a = = t_b だったのか？あるいは本当に t_a = t_b が意図した記述なのか？が分からないという理由からである。

（記述例 3）

```
if ( t_a = t_b ) {
        foo( ) ;
}
```

また、ルール 15.2 は“空でない switch 節は、無条件 break で終了しなければならない”であり、記述例 4 のような記述を禁止している。case 0 には break がないため case0 の場合には関数 func1 と func2 が実行されることになるが、このような場合には break がないことがミスかどうか設計意図の理解が難しいために禁止となる。

（記述例 4）

```
switch( cnd ) {
case 0:
      func1();
case 1:
     func2();
      break;
default:
      func_others();
      break;
}
```

この記述例 4 に関しては、静的なコードチェックツールの lint では、/* FALLTHROUGH */　というコメントで警告を出さないことも可能なので、このコメントをつけた場合には OK とするなどといった議論もあったが、コメントを忘れた場合や制御フローを変更したときにコメントを削除し忘れる恐れなどがあり、いくつかの反論があったものの準拠することとした。このような判断を実施し、この視点でも逸脱をしない方向で対応することにした。

3.3 マイコン/コンパイラ依存の記述

マイコンやコンパイラに依存する部分の記述については、議論がいくつかあり対処法も含めて検討を実施した。この視点での問題は基本的にはマイコンを含めたコンパイラの未定義/未規定から発生しており、移植性などで大きな問題となる可能性がある。

基本的にはルールに準拠するが、以下の項目については検討を実施した。

- 翻訳限界（Translation limits）
- 演算に関する部分

3.4 翻訳限界

MISRA-C では C 言語の規格 C90 を基本としているため、コンパイラが翻訳する際に最低限守るべき基準がある。そのため決められた基準を超えるものは、移植時などに問題が発生する可能性があることになる。例えば、変数の 31 文字制限や、入れ子の段数、括弧の数と言った 22 の項目が C 言語では翻訳限界として規定されている。注意する点としては、MISRA-C では“識別子が 31 文字を超える特徴に依存してはならない”となっているが、C90 では、外部識別子は 6 文字の特徴となっているため、コンパイラの動作確認が必要となっている。

現状の多くのコンパイラの制限からすると、C90 の限界

Vol.2012-SE-176 No.4
Vol.2012-EMB-25 No.4
2012/5/21

値は非常に少ない限界値となっており、大規模なプログラム開発をする上では厳しい場合がある。例えば、C90 の次の規格となる C99 はこの翻訳限界が大きくなっている（表 1）。

表 1　代表的な翻訳限界
Table 1　Typical translation limits

	C90	C99
#if 文のネストの数	8	63
#include のネストの数	8	15
1 つのブロック中での識別子の数	127	511
外部変数の有効文字数	6	31

ただし、古いバージョンのコンパイラを使用する場合などを考えると、一律で準拠しないとも決めることができない。そのため運用としては、遵守できる場合には遵守するが、できない場合にどうするかについては個別に決定することにした。

3.5 演算に関する部分

演算に関してはいくつか移植性などで問題になる点が MISRA-C のルールでは挙げられている。

- 浮動小数点（ルール 1.5）
- 除算（ルール 3.3）
- 符号付右シフト（ルール 12.7）
- 負数のビット演算（ルール 12.7）

現状の業務では浮動小数点演算に関しては必要性が薄いため、今回の検討からははずし、残りの対応について検討を実施した。

4. 演算に関する考察

4.1 問題となる点

3.5 で挙げられた演算(浮動小数点演算を除いて)についてそれぞれ問題となる点を整理する。

- 除算

一方のオペランドが負の値を持つ場合、割り切れない場合の商の値を切り上げ/切り捨ての実装方法があり、この差により予期しない動作をする可能性があることが問題となる。例えば、-5/3 の結果、商が-1 で余りが-2 なのか、商が-2 で余りが 1 になるのかは実装依存となっている。

- 符号付き右シフト

C 言語の規格では符号付オペランドの右シフト演算は処理系依存になっている。

- 負数のビット演算

C 言語では負数の表現形式を規定されていないため、負数のビット演算結果もマイコン/処理系依存となっている。ここでビット演算としているものは、～、＜＜、＞＞、&、｜、＾の 6 つの演算子である。

4.2 現状の調査および方針決定

先ほど挙げた問題に対して、現状組込みで用いられているマイコン/コンパイラについて調査を実施する必要があると考えた。そこで、我々がターゲットとして多く利用している V850, SH2A を中心に調査を行った。また、参考として x86 についても動作を調査した。調査の方法としては、C 言語で記述しその結果を確認した。ここで注意する点は各演算子がマイコンの命令セットに 1 対 1 でマッピングされない場合もあり、複数の命令で実行される場合などがある。この時、マイコンとコンパイラはセットで評価するためにここで評価した組み合わせは以下の通りである。

V850　GHS 社　MULTI 3.3.2
SH2A　ルネサス社　HEW 4.09.00.007
x86 GCC 3.4.4

除算に関しては商を切り上げるのか/切り捨てるのかという実装の違いがあり、特に商が負の数の場合は余りがマイナスなのかプラスになるかで差が出る。実際に調査した結果の一部を表 2 に示す。結果からは、これらに関しては同一の演算結果となった。

表 2　符号付き除算の演算結果
Table 2　Result of the signed division

	V850		SH2A		x86	
演算	商	余	商	余	商	余
-5/3	-1	-2	-1	-2	-1	-2
5/-3	-1	2	-1	2	-1	2
-5/-3	1	-2	1	-2	1	-2

符号付き数に対してのシフト演算に関しての調査した結果を表 3 に示す。シフト量を決定する右オペランドに正の整数を入れた場合には異なった CPU 間で同一の値を得ることができたが、シフト量を決定する右オペランドに負数を入れた場合については、演算結果に差異が現れた。

表 3　符号付数のシフトの演算結果
Table 3　Result of the signed shift

演算	V850	SH2A	x86
-15>>3	-2	-2	-2
15<<-3	0	1	0
-15<<-3	0	-2	0

符号付き数に対して除算とシフトの演算結果について

調査した結果の一部を表 4 に示す。正の数の場合は除算とシフトでの演算結果が一致するが、負の数の場合には一致しない。

表 4　符号付きシフトと除算の演算結果
Table 4　Result of signed shift and signed division

演算	V850	SH2A	x86
-15>>3	-2	-2	-2
-15/8	-1	-1	-1
13>>3	1	1	1
13/8	1	1	1

ビット演算子について調査した結果の一部を表 5 に示す。今回調査した CPU では負数は 2 の補数で表現されており、符号桁（最上位ビット）も含めてビット演算されることを確認し、その結果も同一になることを確認した。

表 5　符号付きビット演算結果
Table 5　Result of the signed bitwise operations

演算		V850	SH2A	x86
&	-5&3	3	3	3
	5&-3	5	5	5
	-5&-3	-7	-7	-7
|	-5|3	-5	-5	-5
	5|-3	-3	-3	-3
	-5|-3	-1	-1	-1
~	~-5	4	4	4
^	-5^3	-8	-8	-8
	5^-3	-8	-8	-8
	-5^-3	6	6	6

演算結果とその時のフラグ動作について V850[3], SH2A[4]についてマニュアルで調査した結果を表 6 に示す。V850 に関しては算術左シフト命令、SH2A に関しては剰余演算を直接持たずに、コンパイラがそれぞれに演算に対応するコードを生成していた。

基本的には SH2A はキャリーなどのフラグ情報は命令種で使い分けることが多く、V850 は演算するとフラグ情報を更新する（表 6）。

表 6：V850 と SH2A の命令とフラグへの影響
Table 6　instructions of SH2A and V850 and influence on flags

演算子	V850	SH2A
&	AND フラグあり	AND　フラグなし
|	OR フラグあり	OR　フラグなし
^	XOR フラグあり	XOR　フラグなし
～	NOT フラグあり	NOT　フラグなし
算術＞＞	SAR フラグあり	SHAD　フラグなし SHAR　フラグあり*
算術＜＜	—	SHAD　フラグなし SHAL　フラグあり*
論理＞＞	SHR フラグあり	SHLD　フラグなし SHLR　フラグあり*
論理＜＜	SHL　フラグあり	SHLD　フラグなし SHLL　フラグあり*
／	DIV　フラグあり	DIV　フラグなし
%	DIV　フラグあり	－

* 1 ビットシフト

演算結果については、符号付きのシフトで右オペランドに負数を入れた場合に結果が異なるが、それ以外の場合については結果が同じであった。ここで、MISRA-C のルール 12.8 に“シフト演算子の右側のオペランドの値は、0 以上、かつ、左側のオペランドの潜在側のビット幅未満でなければならない”というルールがある。MISRA-C 準拠ということを考えると、MISRA-C で許されている使用範囲であれば、今回の結果は同じ演算結果であるといえる。

4.3 対応方法の検討

状況としては、除算、符号付右シフトに関しても上記の MISRA-C ルールを準拠すれば差は無く、負数のビット演算に関しては、現状使用しているマイコンでは 2 の補数を使用しており差が無い。この結果から、案１）現在われわれの利用しているマイコンでは差異はなく、演算子についてはこれまで通りの記述をする、案２）今回の調査では差がないが今後使用するマイコンがすべて同様の結果になるとは限らないためそれに対応できる記述をする、という意見があり、どうするかについて検討した。

案１に対する支持も多くあったが、演算子は記述中で数多く使用され、また、符号なし、符号付きの場合がある。今回問題になっているのは符号付きの場合だけであり、grep などのツールでは符号付きの演算だけを抽出することが出来ないといった問題があり、我々としては案２で検討を進めることとした。ただし、実際の運用時には各ソフトの使用範囲などを含めて対応することにした。

案２としての実現案としては、演算子についてライブラリを用意し、ライブラリ側でマイコン/コンパイラに依存する部分を実装し、アプリ側はマイコン/コンパイラに非依存なコードに出来るようにした（図 1）。

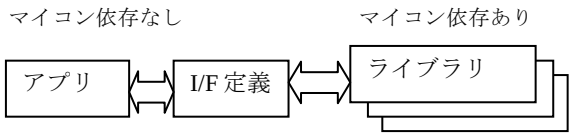

図 1　アプリケーションライブラリの関係
Figure 1　Relationship between applications and library

4.4 仕様策定

ライブラリの仕様として検討したことは、まず負の表現形式としては、現状ターゲットとしている CPU が 2 の補数であり、ライブラリの仕様としては負数は 2 の補数とする。演算結果のフラグなどについては、横並びがうまく取れないこと、および C 言語の演算子の置き換えを考えると C レベルの記述では演算フラグに関する記述をすることができないことから、今回の検討では演算結果のみを保証し、演算フラグ（例：キャリー、オーバーフローなど）についてはサポートしないこととした。

符号付き右シフトに関しては、シフト演算子の代わりに除算を使うプランもあったが結果が異なるパターンがあり、今回は符号付きの右シフトを仕様として定義した。
表 7 に示すライブラリ関数を用意することにした。

表 7　ライブラリ一覧
Table 7　List of library functions

演算子	ライブラリ名	説明
<<	misra_sshal_s32	32bit 符号付左シフト
	misra_sshal_s16	16bit 符号付左シフト
	misra_sshal_s08	8bit 符号付左シフト
>>	misra_sshar_s32	32bit 符号付右シフト
	misra_sshar_s16	16bit 符号付右シフト
	misra_sshar_s08	8bit 符号付右シフト
/	misra_sdiv_s32	32bit 符号付除算
	misra_sdiv_s16	16bit 符号付除算
	misra_sdiv_s08	8bit 符号付除算
%	misra_smod_s32	32bit 符号付余算
	misra_smod_s16	16bit 符号付余算
	misra_smod_s08	8bit 符号付余算
&	misra_sand_s32	32bit 符号付論理積
	misra_sand_s16	16bit 符号付論理積
	misra_sand_s08	8bit 符号付論理積
|	misra_sor_s32	32bit 符号付論理和
	misra_sor_s16	16bit 符号付論理和
	misra_sor_s08	8bit 符号付論理和
^	misra_sxor_s32	32bit 符号付排他論理和
	misra_sxor_s16	16bit 符号付排他論理和
	misra_sxor_s08	8bit 符号付排他論理和
~	misra_scomp_s32	32bit 符号付否定
	misra_scomp_s16	16bit 符号付否定
	misra_scomp_s08	8bit 符号付否定

ビット演算については、負の数は 2 の補数で表現し、符号ビットなど関係なくビット演算を実施する。
除算は、負の数を 2 の補数で表現し、商を切り上げる。
右シフトは符号桁を保持しながら、データの右シフトを実施する。左シフトは空いたビットには 0 詰めを実施し、符号桁は関係なく単純に左シフトを実施する。
この仕様を明確に定義することで、マイコン/コンパイラを変更した場合にテストする項目も明確となり、移植なども容易となる。

4.5 実装の検討

ライブラリの実装についてはいくつか検討する項目がある。ここで議論している演算子の処理系依存という点を考えると、マイコンおよびコンパイラでどのように対応するか決まっていないことにより問題となる状況であり、逆にマイコンとコンパイラが決まれば動作は決まることを考えると、ライブラリ自体の MISRA-C 対応ではマイコン/コンパイラに関して固定で考えることができる(そのかわり、マイコン/コンパイラが変われば最低でもテストが必要となる)。そのため、コンパイラを 4.2 に固定して検討を実施した。

演算の記述については 4.4 で定義した仕様であれば、V850 および SH2A の場合は C 言語の演算子で記述することで仕様を満たす結果を得ることができる。

ここでは、そのライブラリの実装方法としてまずどのような実装にするかを検討した。

- 関数
- マクロ関数
- インライン関数

まずは、演算子の場合を含めて上記の 3 つの場合のコンパイル結果を確認した。
ここでは以下の記述例 5 をベースとして、上記の 3 つの実装法により何が異なるかについて調査を実施した。

(記述例 5)
```
int foo0(    ) {
    s32_gc = s32_ga >> s32_gb;
}
```

V850 に対しては関数化を記述例 6、マクロ関数化を記述例 7, インライン関数を記述例 8 として作成した。

Vol.2012-SE-176 No.4
Vol.2012-EMB-25 No.4
2012/5/21

(記述例 6)

```
s32 misra_sshar_s32(s32 dat,s32 w) {
   return dat >> w;
}
int foo2() {
    s32_gc = misra_sshar_s32(s32_ga, s32_gb);
}
```

（記述例 7）

```
#define dmisra_sshar(a,b) (a>>b)
int foo1() {
    s32_gc = dmisra_sshar(s32_ga, s32_gb);
}
```

（記述例 8）

```
__inline
s32 imisra_sshar_s32(s32 dat,s32 w) {
   return dat >> w;
}
int foo3() {
    s32_gc = imisra_sshar_s32(s32_ga, s32_gb);
}
```

上記を V850 用 MULTI でコンパイルした結果は以下の図 2 ようになる。演算子で記述した関数 foo0 に対して、マクロ関数として記述した関数 foo2 はまったく同一のコードを生成している。関数で実装した場合にはリンクポインタの保存や関数呼び出しの処理が加わっているのがわかる。今回で興味深い点は、インライン関数の場合はレジスタの割り当てが変わっているが、その他の部分は同じであるということである。インライン関数は処理負荷としてはマクロ関数と同じであるが、レジスタまでは一致しない。

```
000008f0 <_foo0>:
 8f0:  movhi          -130, r0, r15
 8f4:  movhi          -130, r0, r14
 8f8:  ld.w 4[r14], r14
 8fc:  ld.w 0[r15], r15
 900:  movhi          -130, r0, r1
 904:  sar   r14, r15
 908:  st.w r15, 8[r1]
 90c:  jmp [lp]

0000092c <_foo2>:
 92c:  prepare        {lp}, 0
 930:  movhi          -130, r0, r6
 934:  movhi          -130, r0, r7
 938:  ld.w 4[r7], r7
 93c:  ld.w 0[r6], r6
 940:  jarl  8e8 <_misra_sshar_s32>, lp
 944:  movhi          -130, r0, r1
 948:  st.w r10, 8[r1]
 94c:  dispose        0, {lp}, lp

0000090e <_foo1>:
 90e:  movhi          -130, r0, r15
 912:  movhi          -130, r0, r14
 916:  ld.w 4[r14], r14
 91a:  ld.w 0[r15], r15
 91e:  movhi          -130, r0, r1
 922:  sar   r14, r15
 926:  st.w r15, 8[r1]
 92a:  jmp [lp]

00000950 <_foo3>:
 950:  movhi          -130, r0, r16
 954:  movhi          -130, r0, r15
 958:  ld.w 4[r15], r15
 95c:  ld.w 0[r16], r16
 960:  movhi          -130, r0, r1
 964:  sar   r15, r16
 968:  st.w r16, 8[r1]
 96c:  jmp [lp]
```

図 2：V850 のコンパイル結果

Figure 2　Result of the compilation (V850)

SH2A に対しても V850 と同様に記述例 5 をベースに、関数化を記述例 9、マクロ関数化を記述例 10, インライン関数を記述例 11 として作成した。

（記述 9）

```
s32 misra_sshar_s32(s32 dat,s32 w) {
   return dat >> w;
}
int foo2() {
    s32_gc = misra_sshar_s32(s32_ga, s32_gb);
}
```

（記述 10）

```
#define dmisra_sshar(a,b) (a>>b)
int foo1() {
    s32_gc = dmisra_sshar(s32_ga, s32_gb);
}
```

（記述 11）

```
#pragma inline( imisra_sshar_s32 )
```

```
s32 imisra_sshar_s32(s32 dat,s32 w) {
   return dat >> w;
}
int foo3() {
    s32_gc = imisra_sshar_s32(s32_ga, s32_gb);
}
```

上記を SH2A の HEW を使って、コンパイル結果を確認した（図 3）。V850 の場合と同様に、演算子で記述した場合、マクロ関数、インライン関数を使用した場合は同一のコードを生成し、関数の場合はプロシージャレジスタ PR の更新および関数呼び出しが追加されている。

```
_foo0:
00001020 MOV.L      @(H'00BC:8,PC),R1
00001022 MOV.L      @R1,R2
00001024 MOV.L      @(H'00BC:8,PC),R6
00001026 MOV.L      @R6,R5
00001028 NEG        R2,R4
0000102A MOV.L      @(H'00BC:8,PC),R7
0000102C SHAD       R4,R5
0000102E RTS
00001030 MOV.L      R5,@R7

_foo2:
00001044 STS.L      PR,@-R15
00001046 MOV.L      @(H'0098:8,PC),R1
00001048 MOV.L      @(H'0098:8,PC),R7
0000104A MOV.L      @R1,R5
0000104C BSR        @_misra_sshar_s32:12
0000104E MOV.L      @R7,R4
00001050 MOV.L      @(H'0094:8,PC),R6
00001052 LDS.L      @R15+,PR
00001054 RTS
00001056 MOV.L      R0,@R6

_foo1:
00001032 MOV.L      @(H'00AC:8,PC),R1
00001034 MOV.L      @R1,R2
00001036 MOV.L      @(H'00AC:8,PC),R6
00001038 MOV.L      @R6,R5
0000103A NEG        R2,R4
0000103C MOV.L      @(H'00A8:8,PC),R7
0000103E SHAD       R4,R5
00001040 RTS
00001042 MOV.L      R5,@R7

_foo3:
00001058 MOV.L      @(H'0084:8,PC),R1
0000105A MOV.L      @R1,R2
0000105C MOV.L      @(H'0084:8,PC),R6
0000105E MOV.L      @R6,R5
00001060 NEG        R2,R4
00001062 MOV.L      @(H'0084:8,PC),R7
00001064 SHAD       R4,R5
00001066 RTS
00001068 MOV.L      R5,@R7
```

図 3： SH2A のコンパイル結果
Figure 3 Result of the compilation (SH2A)

今回 2 つの CPU に対して確認した結果、マクロ関数またはインライン関数を使って実装することで、効率よく実装することが可能なことが分かった。ただし、マクロ関数は型チェックが出来ない問題があるため、インライン関数を使用することにした。

インライン関数の中で演算処理の記述に関しては、今回の V850 および SH2A では C 言語の演算子を記述したが、これは仕様を満たす形の実装形態の 1 つとして選択したためである。今回の例では C 言語で記述することで仕様通りに実装できただけで、場合によってはアセンブラなどで記述する必要があると考えている。

また、コンパイラによってはインライン関数が使用できない場合などがあるが、その時には関数としてライブラリを実装すれば、インタフェースとしては同じ形となるため、関数呼び出しのオーバーヘッドはあるが機能的には問題なく動作する。

4.6 既存ソフトの MISRA-C 対応時のテスト

今回の演算ライブラリを使用した結果、命令、レジスタ、定数（アドレスオフセット、即値など）がすべて演算子を使った場合と一致する場合と命令は一致しているがレジスタやアドレスオフセットが違う場合の 2 つのケースについて考える。

1 つ目のケースの演算子を使った記述と生成されたコードが同一であれば、既存のソフトウェアを MISRA-C 対応した際にテストする必要性はない（当然、既存のコードの時にはテストを十分しているという前提）。これにより、既存のコードを修正/追加して MISRA-C 対応する必要がある場合に、すべてのテストをやり直す必要はない。特に、単体テストで生成されたコードが同一であれば、テスト不要とできる。

2 つ目のケースである命令は一致しているがレジスタやアドレスオフセットが違う場合について検討する。
V850 のコンパイル結果では、命令はすべて一致しているがレジスタが異なっている。レジスタについては、コンパイラでは ABI が決まっており、関数側で自由につかってよい

（関数内で値を変更しても、関数終了時に復帰する必要ない）スクラッチレジスタが違う時に、全体の制御/データフローが一致していれば同一とみなすことが出来る。このフロー一致があれば、テスト不要とできる。

SH2A のコンパイル結果では MOV.L 命令でのオフセットが異なっているが、これは PC 相対のためプログラムのアドレスが異なるためオフセット値も異なる結果になっている。

アドレスとオフセットを加算すると同一のアドレスをアクセスしていることが分かった(MOV.L 命令は下位 2 ビットは 0 固定のため 0xFFFFFFC でマスクする)。

(0x00001020 + 0x00BC) & 0xFFFFFFFC = 0x000010DC
(0x00001032 + 0x00AC) & 0xFFFFFFFC = 0x000010DC
(0x00001058 + 0x0084) & 0xFFFFFFFC = 0x000010DC

これにより分かったこととしては、コンパイル結果が MISRA-C 対応前とまったく同じ場合だけでなく出てくる命令のシーケンスは同じでオフセットやレジスタのみが異なる場合でも、テスト不要なケースが存在するということが分かった。これに対しては簡単なスクリプトで確認することが出来る。

命令が異なる場合でもテスト不要なケースがあるかもしれないが、現状、我々では対応できていない。

4.7 MISRA-C 対応について

今回の演算子に対応することでアプリケーション側は MISRA-C 準拠をすることになる。ライブラリに関しては、仕様定義を実施し、マイコン/コンパイラごとに確認を実施することでルールについては確認できているが、今回の V850、SH2A に関してはルール 12.7 に関しては逸脱となる。ただし、問題になっている演算子を未定義/未既定のまま使用しているわけでなく、仕様/テスト結果なども明確にしているため安全性を保証することが可能と考えている。

また、今回の記述ではインライン関数または関数で実装しているので、呼び出し箇所および定義場所を特定することが容易にできるというメリットもある。演算子で書いた場合には、signed や unsigned を判断する必要があり単純には grep などで検索できない。

5. おわりに

MISRA-C に関しては基本的に遵守する方向で検討を進めたが、C 言語で定義/規定されていない演算については演算ライブラリを用意しマイコン/コンパイラ違いを吸収する方法をとった。

V850 と SH2A に関してはライブラリを実装し、テストにおける考察を行い、MISRA-C 対応前と対応後で完全一致しない場合でも、テスト不要な場合があることを示した。

今回は V850, SH2A のみの調査を実施したが、まだ調査できていない CPU についても継続して調査したい。また、今後、MISRA-C に対して対応方法などについては情報収集を進めていきたい。

参考文献

1) MISRA‐C 研究会: 組込み開発者におくる MISRA‐C:2004―C 言語利用の高信頼化ガイド, 日本規格協会(2006)
2) 自動車技術会 規格委員会: 自動車用 C 言語利用のガイドライン（第 2 版）, 自動車技術会(2006)
3) ユーザーズ・マニュアル V850E2 32 ビット・マイクロプロセッサ・コア アーキテクチャ編 U17135JJ1V1UM00, ルネサスエレクトロニクス 2004
4) SH-2A, SH2A-FPU ユーザーズマニュアル ソフトウェア編, Rev4.00, ルネサスエレクトロニクス 2011