

序説

Atmel® megaAVR® 8ビット マイクロ コントローラは数ある新しい増強の中に加わるハードウェア乗算器を含むAVR® RISCマイクロ コントローラ内の新しいデバイスの系列です。この乗算器は2クロック周期だけを使用して16ビットの結果を与える、2つの8ビット数値の乗算が可能です。乗算器は速度とコード量の不利なく、符号付きと符号なし両方の整数と固定小数点数を扱えます。本資料の最初の項は8ビット演算に関するいくつかの乗算器使用例を与えます。

特徴

- 8ビットと16ビットの実装
- 符号付きと符号なしのルーチン
- 符号付きと符号なしの固定小数点乗算
- 実行可能な例プログラム

目次

序説	1
特徴	1
1. 説明	3
2. 8ビット乗算	4
2.1. 例1 - 基本的な使い方	4
2.2. 例2 - 特殊な状態	4
2.3. 例3 - 乗算/累積操作	4
3. 16ビット乗算	5
3.1. 16×16=16ビット操作	5
3.2. 16×16=24ビット操作	6
3.3. 16×16=32ビット操作	6
3.3.1. 例4 - 16×16=32ビット整数乗算の基本的な使い方	6
3.4. 16ビット乗算/累積操作	6
4. 小数点数の使い方	7
4.1. 例5 - 8×8=16ビット符号付き小数点乗算の基本的な使い方	8
4.2. 例6 - 乗算/累積操作	8
5. 実装に於ける解説	8
6. 改訂履歴	8

1. 説明

乗算器をえるようにAVR命令群に6つの新命令が追加されています。これらの命令は次の通りです。

- MUL – 符号なし整数の乗算
- MULS – 符号付き整数の乗算
- MULSU – 符号付き整数と符号なし整数の乗算
- FMUL – 符号なし固定小数点数の乗算
- FMULS – 符号付き固定小数点数の乗算
- FMULSU – 符号付き固定小数点数と符号なし固定小数点数の乗算

MULSUとFMULSU命令には16ビットオペランドの乗算に対する速度とコード密度の改善が含まれます。2つ目の項は16ビット演算に対して乗算器を効果的に使用する方法の例を示します。

信号処理に対して特別に仕上げられた専用デジタル信号処理器(DSP)にさせる部分は累積器(MAC)です。この部分は論理演算部(ALU)に直接繋がった乗算器と機能的に当価です。megaAVRは同じ累積操作を効率的に実行する能力をAVR系統に与えるように設計されています。従って、この応用記述はMAC動作を実装する例を含みます。

(度々乗算/加算動作として参照される)累積動作には1つの重要な欠点があります。複数の値を1つの結果変数に加算するとき、お互いを或る程度打ち消すために正と負の値を加算する時でも、その限度を越えることで結果変数の危険が明らかになり、例えば値127を含む符号付きバイト変数に1を加算する場合、その結果は+128に代わって-128でしょう。この問題を解決するために度々使用される1つの解決策は小数点数、換言すると1未満且つ-1以上の数値を導入することです。最終項は小数点数の使用に関するいくつかの論点を示します。

新しい乗算命令に加えて、少しの他の追加と改良がmegaAVRプロセッサコアに行われています。特に有用な改良の1つは或るレジスタ対を他のレジスタ対に複写する(レジスタ語複写)、新命令MOVWです。

"AVR201.asm"ファイルは16ビット乗算ルーチンの応用記述ソースコードを含みます。

核となる性能特性と共に全実装の一覧が下表で与えられます。

表1-1. 性能要約

分類	種別	コード量(語)	実行時間(周期)
8ビット×8ビットルーチン	符号なし, 8×8=16ビット乗算	1	2
	符号付き, 8×8=16ビット乗算	1	2
	符号付き/符号なし固定小数点, 8×8=16ビット乗算	1	2
	符号付き固定小数点, 8×8+16=16ビット乗算/累積	3	4
16ビット×16ビットルーチン	符号付き/符号なし, 16×16=16ビット乗算	6	9
	符号なし, 16×16=32ビット乗算	13	17
	符号付き, 16×16=32ビット乗算	15	19
	符号付き, 16×16+32=32ビット乗算/累積	19	23
	符号付き固定小数点, 16×16=32ビット乗算	16	20
	符号付き固定小数点, 16×16+32=32ビット乗算/累積	21	25
	符号なし, 16×16=24ビット乗算	10	14
	符号付き, 16×16=24ビット乗算	10	14
	符号付き, 16×16+24=24ビット乗算/累積	12	16

2. 8ビット乗算

本章内の例が明らかに示すように、ハードウェア乗算器を使用して8ビット乗算を行うのは簡単です。まあ、2つ(または2乗については1つだけ)のレジスタ内にオペランドを格納して乗算命令の1つを実行してみてください。結果がR1:R0レジスタ対に置かれるでしょう。けれども、**MUL**命令だけが使用レジスタ制限を持たないことに注意してください。図2-1は乗算命令の各々に対する有効な(オペランド)レジスタ使用を示します。

2.1. 例1 - 基本的な使い方

最初の例はポートBの入力値を読み、結果をR17:R16レジスタ対に格納する前にその値を定数(5)で乗算するアセンブリ言語コードを示します。

```
IN      R16, PINB      ; ポートB入力値取得
LDI     R17, 5         ; 定数(5)取得
MUL     R16, R17       ; R1:R0=ポートB入力値×5
MOVW   R17:R16, R1:R0 ; 結果をR17:R16レジスタ対に複写
```

MOVW命令の使い方に注意してください。本例は乗算命令の全てについて有効です。

図2-1. 有効なレジスタ使用

MUL	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	R31
MULS																	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	R31
MULSU, FMUL, FMULS, FMLSU																	R16	R17	R18	R19	R20	R21	R22	R23								

2.2. 例2 - 特殊な状態

本例は有効な**MUL**命令のいくつかの特殊な状態を示します。

```
LDS     R0, variableA ; R0にSRAMの変数A値を取得
LDS     R1, variableB ; R1にSRAMの変数B値を取得
MUL     R1, R0         ; R1:R0=変数A×変数B

LDS     R0, variableA ; R0にSRAMの変数A値を取得
MUL     R0, R0         ; R1:R0=(変数A)2
```

例えオペランドが結果のR1:R0レジスタ対に置かれたとしても、R1とR0は最初のクロック周期で取得され、2番目のクロック周期で結果が書き戻されるため、この操作は正しい結果を与えます。

2.3. 例3 - 乗算/累積操作

8ビット乗算の最後の例は乗算/累積操作を示します。一般形は次のように書かれます。

$$c(n) = a(n) \times b + c(n-1)$$

```
; * R17:R16=R18×R19+R17:R16 *

IN      R18, PINB      ; 現在のポートB入力値取得
LDI     R19, b         ; 定数bをR19に取得
MULS   R19, R18       ; R1:R0=ポートB入力値×定数b
ADD    R16, R0         ; R17:R16=R17:R16+R1:R0(累積)
ADC    R17, R1
```

除算/累積操作に関する代表的な応用は、有限インパルス応答(FIR:Finite Impulse Response)と無限インパルス応答(IIR:Infinite Impulse Response)の濾波器(フィルタ)、比例積分微分(PID:Proportional-Integral-Differential)調整器、高速フーリエ変換(FFT:Fast Fourier Transform)です。これらの応用に対しては**FMULS**命令が特に有用です。

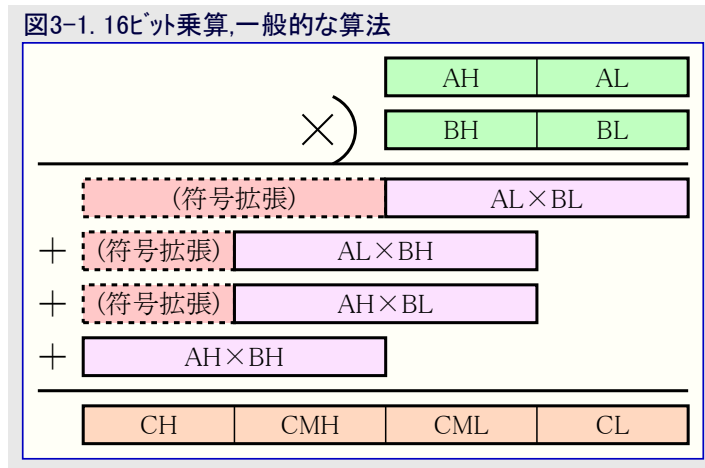
MULS命令の代わりに**FMULS**命令を使用する主な利点は、**FMULS**操作の16ビットの結果が常に(上手く定義された)8ビット形式に近似できることです。これは「**小数点数の使い方**」章で更に論議されます。

3. 16ビット乗算

新しい乗算命令は16ビット乗算を改善するために特別に設計されています。本章は16ビットオペランドでの乗算を行うためのハードウェア乗算器使用に関する解決策を示します。

下図は32ビットの結果を持つ2つの16ビット数値の乗算($A \times B = C$)に対する一般的な算法を図式的に図解します。AHはオペランドAの上位バイト、ALが下位バイトを意味します。CMHは結果Cの中間の上位バイト、CMLが中間の下位バイトを意味します。同様の表記法が残りのバイトについても使用されます。

この算法は基本的に全て乗算です。部分的な16ビットの結果の全てがシフトされ、そして共に加算されます。符号拡張は符号付き数値にだけ必要ですが、符号なし数値に対してキャリーの伝播が未だ行われなければならないことに注意してください。



3.1. $16 \times 16 = 16$ ビット操作

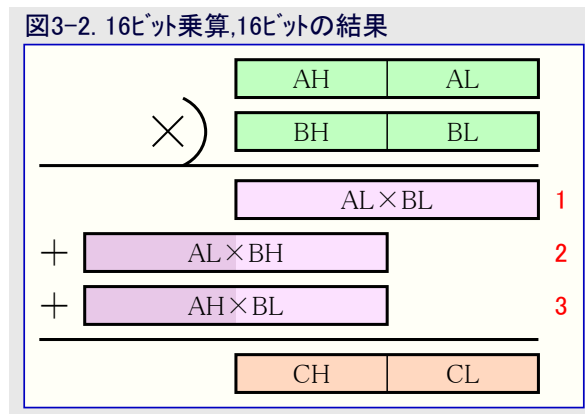
例えば符号なし乗算命令(MUL)だけが必要とされたとしても、この操作は符号付きと符号なしの両方の数値について有効です。これは下図で図解されます。数学的な説明が以下で与えられます。

AとBが正の数値、または少なくともそれらの1つが0のとき、 $A \times B$ の積Cが符号なし数値として使用された場合に 2^{16} 未満なら、積が符号付き数値として使用される場合に 2^{15} 未満なら、この算法は明らかに間違いありません。

両因数が負のとき、2の補数表記、 $A = 2^{16} - |A|$ と $B = 2^{16} - |B|$ が用いられます。

$$C = A \times B = (2^{16} - |A|) \times (2^{16} - |B|) = |A \times B| + 2^{32} - 2^{16} \times (|A| + |B|)$$

ここで我々は16LSB(下位16ビット)だけに係り、この総和の最後に残った部分は破棄され、 $|A \times B| = C$ の(正しい)結果を得ます。



1つの因数が負で他の1つの因数が正、例えばAが負でBが正の時は以下です。

$$C = A \times B = (2^{16} - |A|) \times |B| = (2^{16} \times |B|) - |A \times B| = (2^{16} - |A \times B|) + 2^{16} \times (|B| - 1)$$

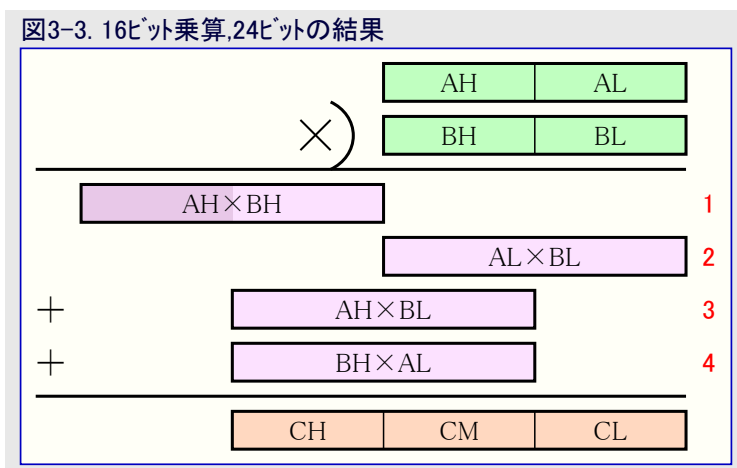
MSBは破棄され、正しい2の補数表記の結果は $C = 2^{16} - |A \times B|$ です。

積は符号なし数値が使用される場合に $0 \leq C \leq 2^{16} - 1$ の範囲、符号付き数値が使用される場合に $-2^{15} \leq C \leq 2^{15} - 1$ の範囲でなければなりません。

C言語で整数乗算を行うとき、これがどう行われるかです。この算法は32ビットの結果を持つ32ビット乗算を行うことで説明できます。

3.2. 16×16=24ビット操作

このルーチンの機能は下図で図解されます。24ビット版の乗算ルーチンについては、その結果がR18:R17:R16レジスタで示されます。符号なし乗算使用時にA×Bの積Cが 2^{24} 未満なら、符号付き乗算使用時に $\pm 2^{23}$ 未満なら、この算法は正しい結果を与えます。

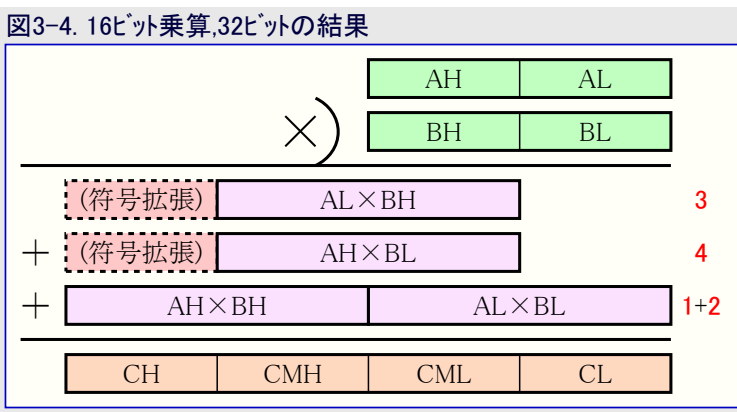


3.3. 16×16=32ビット操作

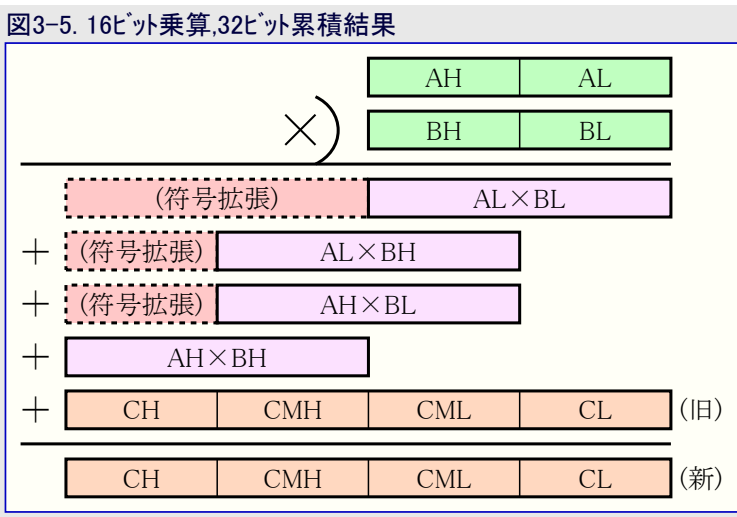
3.3.1. 例4 - 16×16=32ビット整数乗算の基本的な使い方

以下は16×16=32乗算サブルーチンの呼び出し法の例です。これは下図でも図解されます。

```
LDI R23, HIGH(672) ; R23:R22に数値672を設定
LDI R22, LOW(672)
LDI R21, HIGH(1844) ; R21:R20に数値1844を設定
LDI R20, LOW(184)
CALL mul16x16_32 ; 16×16=32ビット乗算ルーチン呼び出し
```



3.4. 16ビット乗算/累積操作



4. 小数点数の使い方

符号なし8ビット小数点数は(0以上2未満)範囲内の数値が許される形式、ビット6~0が小数を表し、ビット7は整数部(0または1)を表す、換言すると1.7形式を使用します。2バイトの結果の上位バイトが(2.6形式に代えて)オペラントと同じ1.7形式を持つように結果が1ビット左にシフトされるのを除いて、**FMUL**命令は**MUL**命令と同じ操作を実行します。積が2以上の場合に結果が正しくなくなることに注意してください。

小数点数の形式を完全に理解するのに整数形式との比較が有用です。下表はこの2つの符号なし数値形式を図解します。符号付き整数のように、符号付き小数点数は良く知っている2の補数形式を使用します。-1以上1未満の範囲の数値がこの形式を使用して表現できます。

バイト'1011 0010'が符号なし整数として解釈される場合、それは $128+32+16+2=178$ と解釈されるでしょう。これに反して、符号なし小数点数として解釈される場合は $1+0.25+0.125+0.015625=1.390625$ と解釈されるでしょう。このバイトが符号付き数値と仮定されるなら、 $178-256=122$ (整数)または $1.390625-2=-0.609375$ (小数点数)と解釈されるでしょう。

表4-1. 整数と小数点の形式比較

ビット番号		7	6	5	4	3	2	1	0
ビットの意味	符号なし整数	$2^7=128$	$2^6=64$	$2^5=32$	$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
	符号なし小数点数	$2^0=1$	$2^{-1}=0.5$	$2^{-2}=0.25$	$2^{-3}=0.125$	$2^{-4}=0.0625$	$2^{-5}=0.03125$	$2^{-6}=0.012625$	$2^{-7}=0.0078125$

FMUL, **FMULS**, **FMULSU**命令は**MUL**, **MULS**, **MULSU**命令より僅かに複雑でしょう。とは言え、1つの潜在的な問題は簡単な方法で適切な値を小数点変数に割り当てることです。小数0.75($=0.5+0.25$)は、例えば8ビットが使用される場合に'0100 0000'になるでしょう。

0から2未満の範囲で正の小数点数(例えば1.8125)をAVRで使用する形式に変換するには、例で図解される以下の算法が使用されるべきです。

数値内に"1"がありますか?

はい、1.8125は1以上です。

バイトは今や'1xxx xxxx'です。

残りに"0.5"はありますか?

$$0.8125 \div 0.5 = 1.625$$

はい、1.625は1以上です。

バイトは今や'11xx xxxx'です。

残りに"0.25"はありますか?

$$0.625 \div 0.5 = 1.25$$

はい、1.25は1以上です。

バイトは今や'111x xxxx'です。

残りに"0.125"はありますか?

$$0.25 \div 0.5 = 0.5$$

いいえ、0.5は1未満です。

バイトは今や'1110 xxxx'です。

残りに"0.0625"はありますか?

$$0.5 \div 0.5 = 1$$

はい、1は1以上です。

バイトは今や'1110 1xxx'です。

残りが無くなったために残りの3ビットは0で、最終結果は'1110 1000'、そしてこれは $1+0.5+0.25+0.0625=1.8125$ です。

負の小数点数に変換するには、最初に2を数値に加算して、それから既に示されたのと同じ算法を使用してください。

16ビット小数点数は、上位8ビットが8ビット形式と同じ形式を持つ、8ビット小数点数と同様の形式を使用します。下位8ビットは8ビット形式の精度を増すだけです。8ビット形式が 2^{-8} の精度を持つ一方、16ビット形式は 2^{-16} の精度を持ちます。そして更に、32ビット小数点数は16ビット小数点数に対する精度増加です。数値を格納するために追加バイトが使用される時の整数と小数点数間の重要な違い、小数点数使用時に数値の精度が増やされる一方、整数使用時に表現できる数値の範囲が拡張されることに注意してください。

先に言及したように、-1以上1未満の範囲内の符号付き小数点数の使用は整数に対する1つの主要な優位性を持ち、それは-1以上1未満の範囲内の2つの数値の乗算時にその結果が-1以上1以下の範囲内になり、1つの例外付きで結果の近似値(上位側バイト)が因数として同じバイト数で格納できることで、その例外は両因数が-1の時に積が1であるべきなのに、数値1が使用する数値形式で表現できないため、**FMULS**命令はR1:R0内に数値-1を代わりに置きます。従って使用者は**FMULS**命令使用時に少なくともオペラントの1つが-1でないことを保証すべきです。16ビット×16ビット小数点乗算にもこの制限があります。

4.1. 例5 - 8×8=16ビット符号付き小数点乗算の基本的な使い方

本例はポートBの入力値を読み、R17:R16レジスタ対に結果を格納する前に、その値を小数点定数(-0.625)で乗算するアセンブリ言語コードを示します。

```
IN      R16, PINB      ; ポートB入力値取得
LDI     R17, 0b10110000 ; 固定小数点数-0.625値取得
FMULS  R16, R17       ; R1:R0=ポートB入力値×(-0.625)
MOVW   R17:R16, R1:R0 ; 結果をR17:R16レジスタ対に複写
```

FMULS(とFMUL)命令の使い方がMULSとMUL命令の使い方と非常に類似していることに注意してください。

4.2. 例6 - 乗算/累積操作

下の例はA/D変換器からのデータを使用します。A/D変換器はA/D変換結果の形式が2の補数小数点形式に適合するように設定されるべきです。(例えば)ATmega83/163に関しては、これがADMUX入出力レジスタ内の左揃え(ADLAR)ビットを設定(1)し、差動チャンネルが使用されることを意味します(A/D変換の結果は1で正規化されます)。

```
LDI     R23, 0b01100010 ; 固定小数点数0.771484375の上位バイト値取得
LDI     R22, 0b11000000 ; 固定小数点数0.771484375の下位バイト値取得
IN      R20, ADCL       ; A/D変換結果下位バイト値取得
IN      R21, ADCH       ; A/D変換結果上位バイト値取得
CALL    fmac16x16_32    ; 符号付き小数点乗算/累積ルーチン呼び出し
```

A/D変換結果と0.771484375の乗算結果でR19:R18:R17:R16レジスタが増加されるでしょう。本例ではA/D変換の結果が符号付き固定小数点数として扱われます。それを符号付き整数として扱い、“fmac16x16_32”に代えて“mac16x16_32”を呼び出すこともできます。この場合は0.771484375が整数で置き換えられるべきです。

5. 実装に於ける解説

ここで実装される全ての16×16=32ビット関数は、正に“キャリーと共に加算”(ADC)と“ボローと共に減算”(SBC)の操作での替え玉レジスタとして使用されるR2レジスタの解除によって始まります。これらの(関数)操作はR2レジスタ内容を変えません。R2レジスタがコード内の何処か別の処で使用されないなら、それらの関数が呼び出される度毎にR2レジスタを解除する必要はありませんが、関数の1つを最初に呼び出す前に一度だけ必要です。

6. 改訂履歴

資料改訂	日付	注釈
1631A	-	初版資料公開
1631B	-	-
1631C	2002年6月	-
1631D	2016年10月	新雛形

Atmel®, Atmelロゴとそれらの組み合わせ、Enabling Unlimited Possibilities®, AVR®, megAVR®とその他は米国及び他の国に於けるAtmel Corporationの登録商標または商標です。他の用語と製品名は一般的に他の商標です。

お断り: 本資料内の情報はAtmel製品と関連して提供されています。本資料またはAtmel製品の販売と関連して承諾される何れの知的所有権も禁反言あるいはその逆によって明示的または暗示的に承諾されるものではありません。Atmelのウェブサイトに表示する販売の条件とAtmelの定義での詳しい説明を除いて、商品性、特定目的に関する適合性、または適法性の暗黙保証に制限せず、Atmelはそれらを含むその製品に関連する暗示的、明示的または法令による如何なる保証も否認し、何ら責任がないと認識します。たとえAtmelがそのような損害賠償の可能性を進言されたとしても、本資料を使用できない、または使用以外で発生する(情報の損失、事業中断、または利益と損失に関する制限なしの損害賠償を含み)直接、間接、必然、偶然、特別、または付随して起こる如何なる損害賠償に対しても決してAtmelに責任がないでしょう。Atmelは本資料の内容の正確さまたは完全性に関して断言または保証を行わず、予告なしでいつでも製品内容と仕様の変更を行う権利を保留します。Atmelはここに含まれた情報を更新することに対してどんな公約も行いません。特に別の方法で提供されなければ、Atmel製品は車載応用に対して適当ではなく、使用されるべきではありません。Atmel製品は延命または生命維持を意図した応用での部品としての使用に対して意図、認定、または保証されません。

安全重視、軍用、車載応用のお断り: Atmel製品はAtmelが提供する特別に書かれた承諾を除き、そのような製品の機能不全が著しく人に危害を加えたり死に至らしめることがかなり予期されるどんな応用(“安全重視応用”)に対しても設計されず、またそれらとの接続にも使用されません。安全重視応用は限定なしで、生命維持装置とシステム、核施設と武器システムの操作の装置やシステムを含みます。Atmelによって軍用等級として特に明確に示される以外、Atmel製品は軍用や航空宇宙の応用や環境のために設計も意図もされていません。Atmelによって車載等級として特に明確に示される以外、Atmel製品は車載応用での使用のために設計も意図もされていません。

© HERO 2016.

本応用記述はAtmelのAVR201応用記述(Rev.1631D-10/2016)の翻訳日本語版です。日本語では不自然となる重複する形容表現は省略されている場合があります。日本語では難解となる表現は大幅に意訳されている部分もあります。必要に応じて一部加筆されています。頁割の変更により、原本より頁数が少なくなっています。

必要と思われる部分には()内に英語表記や略称などを残す形で表記しています。

青字の部分はリンクとなっています。一般的に赤字の0,1は論理0,1を表します。その他の赤字は重要な部分を表します。