AVR 8-bit Microcontrollers

AVR201: Using the AVR Hardware Multiplier

APPLICATION NOTE

Features

- 8- and 16-bit implementations
- Signed and unsigned routines
- Fractional signed and unsigned multiply
- Executable example programs

Introduction

The Atmel® megaAVR® 8-bit Microcontrollers is a series of devices in the AVR® RISC Microcontroller family that includes, among other new enhancements, a hardware multiplier. This multiplier is capable of multiplying two 8-bit numbers, giving a 16-bit result using only two clock cycles. The multiplier can handle both signed and unsigned integer and fractional numbers without speed or code size penalty. The first section of this document will give some examples of using the multiplier for 8-bit arithmetic.

Table of Contents

Features 1

Introduction 1

1. Description 3

2. 8-bit Multiplication 5

   2.1. Example 1 – Basic Usage 5

   2.2. Example 2 – Special Cases 6

   2.3. Example 3 – Multiply-accumulate Operation 6

3. 16-bit Multiplication 7

   3.1. 16-bit x 16-bit = 16-bit 16-bit x 16-bit = 16-bit Operation 7

   3.2. 16-bit x 16-bit = 24-bit Operation 8

   3.3. 16-bit x 16-bit = 32-bit Operation 9

      3.3.1. Example 4 – Basic Usage 16-bit x 16-bit = 32-bit Integer Multiply 9

   3.4. 16-bit Multiply-accumulate Operation 10

4. Using Fractional Numbers 11

   4.1. Example 5 – Basic Usage 8-bit x 8-bit = 16-bit Signed Fractional Multiply 12

   4.2. Example 6 – Multiply-accumulate Operation 13

5. Comment on Implementations 14

6. Revision History 15

1. Description

To be able to use the multiplier, six instructions are added to the AVR instruction set. These instructions are:

- MUL, multiplication of unsigned integers
- MULS, multiplication of signed integers
- MULSU, multiplication of a signed integer with an unsigned integer
- FMUL, multiplication of unsigned fractional numbers
- FMULS, multiplication of signed fractional numbers
- FMULSU, multiplication of a signed fractional number and with an unsigned fractional number

The MULSU and FMULSU instructions are included to improve the speed and code density for multiplication of 16-bit operands. The second section will show examples of how to efficiently use the multiplier for 16-bit arithmetic.

The component that makes a dedicated digital signal processor (DSP) specially suitable for signal processing is the Multiply-Accumulate (MAC) unit. This unit is functionally equivalent to a multiplier directly connected to an Arithmetic Logic Unit (ALU). The megaAVR microcontrollers are designed to give the AVR family the ability to effectively perform the same multiply-accumulate operation. This application note will therefore include examples of implementing the MAC operation.

The Multiply-Accumulate operation (sometimes referred to as multiply-add operation) has one critical drawback. When adding multiple values to one result variable, even when adding positive and negative values to some extent cancel each other, the risk of the result variable to overrun its limits becomes evident, i.e., if adding one to a signed byte variable that contains the value +127, the result will be -128 instead of +128. One solution often used to solve this problem is to introduce fractional numbers, i.e., numbers that are less than 1 and greater than or equal to -1. The final section presents some issues regarding the use of fractional numbers.

In addition to the new multiplication instruction, a few other additions and improvements are made to the megaAVR processor core. One improvement that is particularly useful is the instruction MOVW - Copy Register Word, which makes a copy of one register pair into another register pair.

The file “AVR201.asm” contains the application note source code of the 16-bit multiply routines.

A listing of all implementations with key performance specifications is given in the table below.

Table 1-1. Performance Summary

8-bit x 8-bit routines	Word [cycles)
Unsigned multiply 8 x 8 = 16 bits	1 (2)
Signed multiply 8 x 8 = 16 bits	1 (2)
Fractional signed/unsigned multiply 8 x 8 = 16 bits	1 (2)
Fractional signed multiply-accumulate 8 x 8 = 16 bits	3 (4)
16-bit x 16-bit routines	
Signed/unsigned multiply 16 x 16 = 16 bits	6 (9)
Unsigned multiply 16 x 16 = 32 bits	13 (17)
Signed multiply 16 x 16 = 32 bits	15 (19)

8-bit x 8-bit routines	Word [cycles)
Signed multiply-accumulate 16 x 16 = 32 bits	19 (23)
Fractional signed multiply 16 x 16 = 32 bits	16 (20)
Fractional signed multiply-accumulate 16 x 16 = 32 bits	21 (25)
Unsigned multiply 16 x 16 = 24 bits	10 (14)
Signed multiply 16 x 16 = 24 bits	10 (14)
Signed multiply-accumulate 16 x 16 = 24 bits	12 (16)

2. 8-bit Multiplication

Doing an 8-bit multiply using the hardware multiplier is simple, as the examples in this chapter will clearly show. Just load the operands into two registers (or only one for square multiply) and execute one of the multiply instructions. The result will be placed in register pair R0:R1. However, note that only the MUL instruction does not have register usage restrictions. Figure 2-1 shows the valid (operand) register usage for each of the multiply instructions.

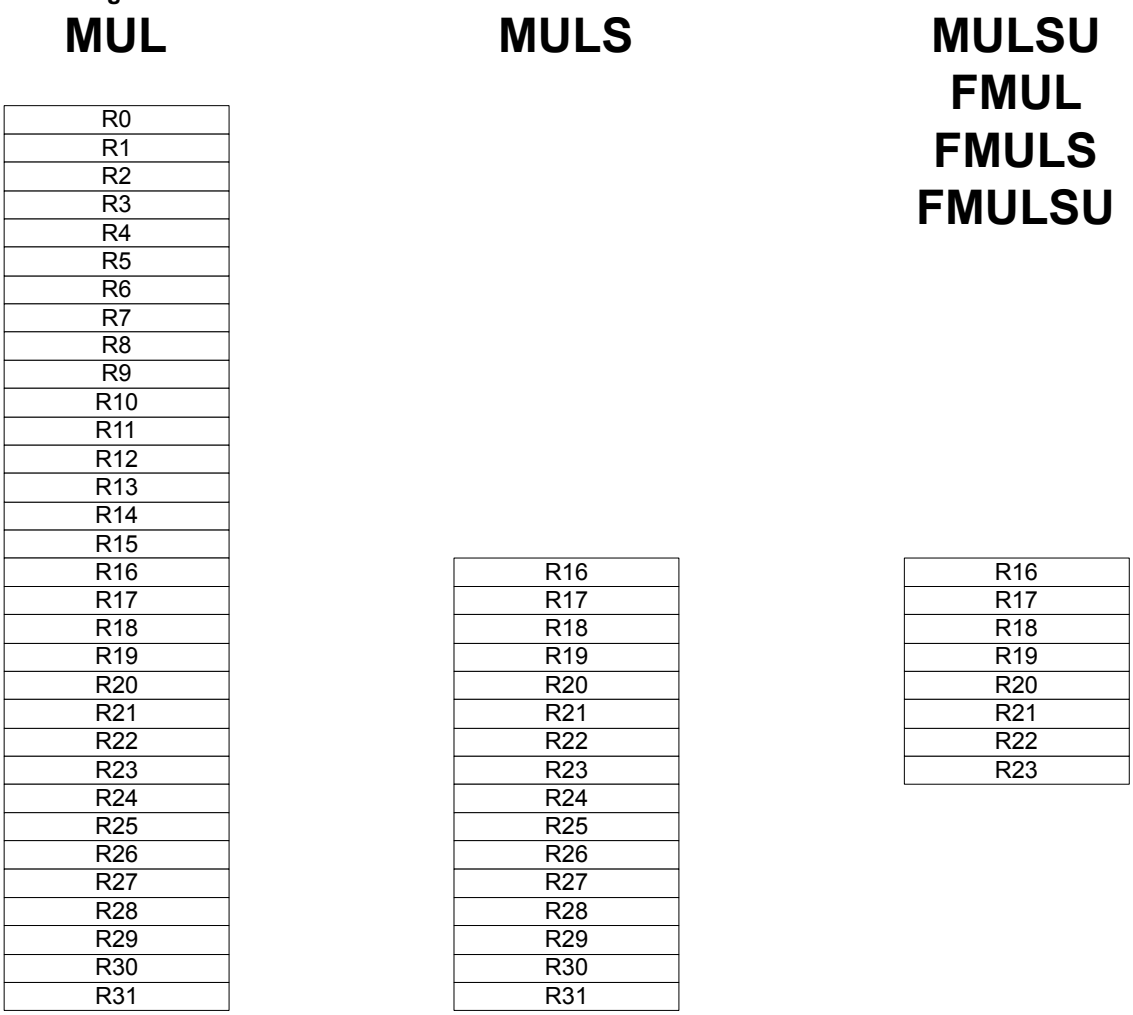
2.1. Example 1 – Basic Usage

The first example shows an assembly code that reads the port B input value and multiplies this value with a constant (5) before storing the result in register pair R17:R16.

```
in r16,PINB ; Read pin values
ldi r17,5 ; Load 5 into r17
mul r16,r17 ; r1:r0 = r17 * r16
movw r17:r16,r1:r0 ; Move the result to the r17:r16 register pair
```

Note the use of the MOVW instruction. This example is valid for all of the multiply instructions.

Figure 2-1. Register Restrictions



2.2. Example 2 – Special Cases

This example shows some special cases of the MUL instruction that are valid.

```
lds r0,variableA ; Load r0 with SRAM variable A
lds r1,variableB ; Load r1 with SRAM variable B
mul r1,r0 ; r1:r0 = variable A * variable B
lds r0,variableA ; Load r0 with SRAM variable A
mul r0,r0 ; r1:r0 = square(variable A)
```

Even though the operand is put in the result register pair R1:R0, the operation gives the correct result since R1 and R0 are fetched in the first clock cycle and the result is stored back in the second clock cycle.

2.3. Example 3 – Multiply-accumulate Operation

The final example of 8-bit multiplication shows a multiply-accumulate operation. The general formula can be written as:

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

```
; r17:r16 = r18 * r19 + r17:r16
in r18,PINB ; Get the current pin value on port B
ldi r19,b ; Load constant b into r19
muls r19,r18 ; r1:r0 = variable A * variable B
add r16,r0 ; r17:r16 += r1:r0
adc r17,r1
```

Typical applications for the multiply-accumulate operation are FIR (Finite Impulse Response) and IIR (Infinite Impulse Response) filters, PID regulators and FFT (Fast Fourier Transform). For these applications the FMULS instruction is particularly useful. The main advantage of using the FMULS instruction instead of the MULS instruction is that the 16-bit result of the FMULS operation always may be approximated to a (welldefined) 8-bit format. This is discussed further in the Using Fractional Numbers section.

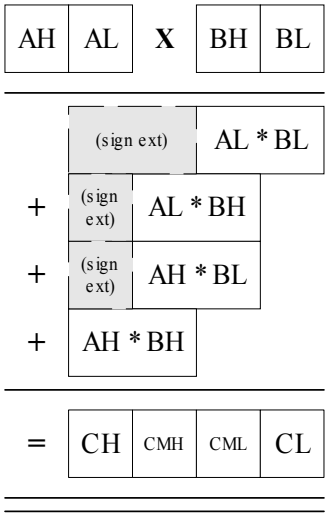
3. 16-bit Multiplication

The new multiply instructions are specifically designed to improve 16-bit multiplication. This chapter presents solutions for using the hardware multiplier to do multiplication with 16-bit operands.

The figure below schematically illustrates the general algorithm for multiplying two 16-bit numbers with a 32-bit result (C = A • B). AH denotes the high byte and AL the low byte of the A operand. CMH denotes the middle high byte and CML the middle low byte of the result C. Equal notations are used for the remaining bytes.

The algorithm is basic for all multiplication. All of the partial 16-bit results are shifted and added together. The sign extension is necessary for signed numbers only, but note that the carry propagation must still be done for unsigned numbers.

Figure 3-1. 16-bit Multiplication, General Algorithm



3.1. 16-bit x 16-bit = 16-bit 16-bit x 16-bit = 16-bit Operation

This operation is valid for both unsigned and signed numbers, even though only the unsigned multiply instruction (MUL) is needed. This is illustrated in the figure below. A mathematical explanation is given:

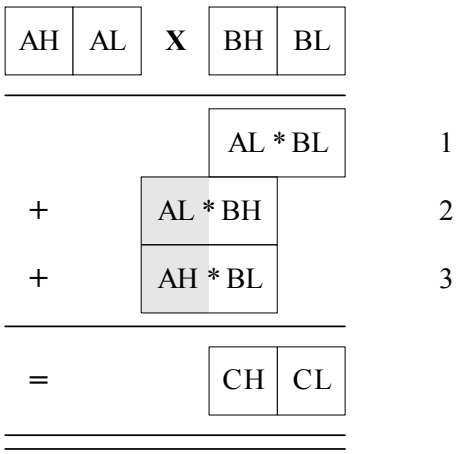
When A and B are positive numbers, or at least one of them is zero, the algorithm is clearly correct, provided that the product C = A • B is less than 2^{16} if the product is to be used as an unsigned number, or less than 2^{15} if the product is to be used as a signed number.

When both factors are negative, the two's complement notation is used; A = 2^{16} - |A| and B = 2^{16} - |B|:

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

Here we are only concerned with the 16 LSBs; the last part of this sum will be discarded and we will get the (correct) result C = |A • B|.

Figure 3-2. 16-bit Multiplication, 16-bit Result



When one factor is negative and one factor is positive, for example, A is negative and B is positive:

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

The MSBs will be discarded and the correct two's complement notation result will be C = 2^{16} - |A • B|.

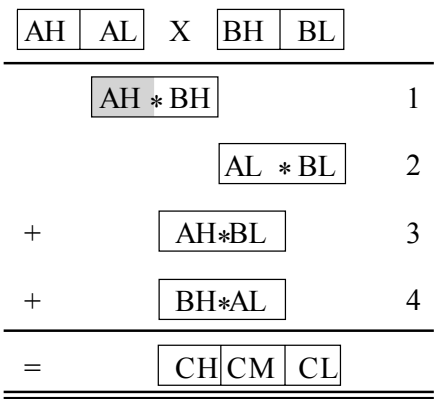
The product must be in the range 0 ≤ C ≤ 2^{16} - 1 if unsigned numbers are used, and in the range -2^{15} ≤ C ≤ 2^{15} - 1 if signed numbers are used.

When doing integer multiplication in C language, this is how it is done. The algorithm can be expanded to do 32-bit multiplication with 32-bit result.

3.2. 16-bit x 16-bit = 24-bit Operation

The routine's functionality is illustrated in the figure below. For the 24-bit version of the multiplication routines, the result is present in registers r18:r17:r16. The algorithm gives correct results provided that the product C = A • B is less than 2^{24} when using unsigned multiplication, and less than ±2^{23} when using signed multiplication.

Figure 3-3. 16-bit Multiplication, 24-bit Result



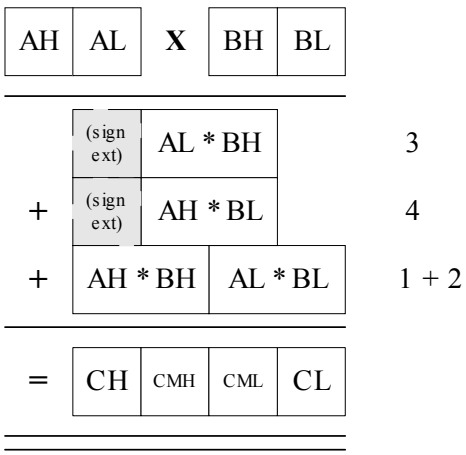
3.3. 16-bit x 16-bit = 32-bit Operation

3.3.1. Example 4 – Basic Usage 16-bit x 16-bit = 32-bit Integer Multiply

Below is an example of how to call the 16 x 16 = 32 multiply subroutine. This is also illustrated in the figure below.

```
ldi R23,HIGH(672)
ldi R22,LOW(672) ; Load the number 672 into r23:r22
ldi R21,HIGH(1844)
ldi R20,LOW(184) ; Load the number 1844 into r21:r20
call mul16x16_32 ; Call 16bits x 16bits = 32bits multiply routine
```

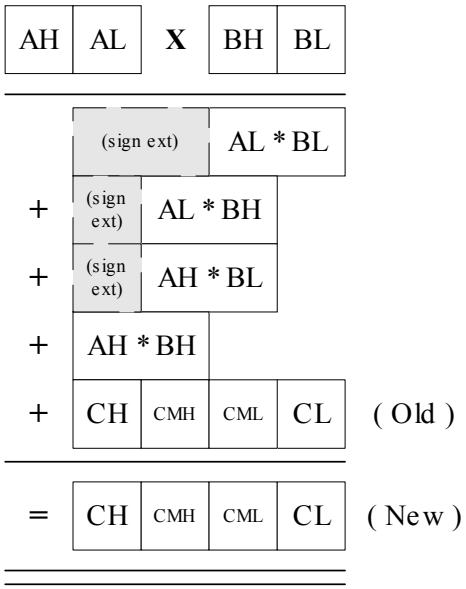
Figure 3-4. 16-bit Multiplication, 32-bit Result



The 32-bit result of the unsigned multiplication of 672 and 1844 will now be in the registers R19:R18:R17:R16. If “muls16x16_32” is called instead of “mul16x16_32”, a signed multiplication will be executed. If “mul16x16_16” is called, the result will be only 16 bits long and will be stored in the register pair R17:R16. In this example, the 16-bit result will not be correct.

3.4. 16-bit Multiply-accumulate Operation

Figure 3-5. 16-bit Multiplication, 32-bit Accumulated Result



4. Using Fractional Numbers

Unsigned 8-bit fractional numbers use a format where numbers in the range [0, 2> are allowed. Bits 6 - 0 represent the fraction and bit 7 represents the integer part (0 or 1), i.e., a 1.7 format. The FMUL instruction performs the same operation as the MUL instruction, except that the result is left-shifted 1-bit so that the high byte of the 2-byte result will have the same 1.7 format as the operands (instead of a 2.6 format). Note that if the product is equal to or higher than 2, the result will not be correct.

To fully understand the format of the fractional numbers, a comparison with the integer number format is useful: The table below illustrates the two 8-bit unsigned numbers formats. Signed fractional numbers, like signed integers, use the familiar two's complement format. Numbers in the range [-1, 1> may be represented using this format.

If the byte "1011 0010" is interpreted as an unsigned integer, it will be interpreted as 128 + 32 + 16 + 2 = 178. On the other hand, if it is interpreted as an unsigned fractional number, it will be interpreted as 1 + 0.25 + 0.125 + 0.015625 = 1.390625. If the byte is assumed to be a signed number, it will be interpreted as 178 - 256 = -122 (integer) or as 1.390625 - 2 = -0.609375 (fractional number).

Table 4-1. Comparison of Integer and Fractional Formats

Bit number	7	6	5	4
Unsigned integer bit significance	2^7=128	2^6=64	2^5=32	2^4=16
Unsigned fractional number bit significance	2^0=1	2^{-1}=0.5	2^{-2}=0.25	2^{-3}=0.125

Bit number	3	2	1	|
Unsigned integer bit significance	2^3=8	2^2=4	2^1=2	2^0=1
Unsigned fractional number bit significance	2^{-4}=0.0625	2^{-5}=0.03125	2^{-6}=0.015625	2^{-7}=0.0078125

Using the FMUL, FMULS, and FMULSU instructions should not be more complex than the MUL, MULS, and MULSU instructions. However, one potential problem is to assign fractional variables right values in a simple way. The fraction 0.75 (= 0.5 + 0.25) will, for example, be "0110 0000" if eight bits are used.

To convert a positive fractional number in the range [0, 2> (for example 1.8125) to the format used in the AVR, the following algorithm, illustrated by an example, should be used:

Is there a "1" in the number?

Yes, 1.8125 is higher than or equal to 1. Byte is now "1xxx xxxx"

Is there a "0.5" in the rest?

0.8125/0.5 = 1.625

Yes, 1.625 is higher than or equal to 1.

Byte is now "11xx xxxx"

Is there a "0.25" in the rest?

0.625/0.5 = 1.25

Yes, 1.25 is higher than or equal to 1.

Byte is now “111x xxxx”

Is there a “0.125” in the rest?

0.25/0.5 = 0.5

No, 0.5 is lower than 1.

Byte is now “1110 xxxx”

Is there a “0.0625” in the rest?

0.5/0.5 = 1

Yes, 1 is higher than or equal to 1.

Byte is now “1110 1xxx”

Since we do not have a rest, the remaining three bits will be zero, and the final result is “1110 1000”, which is 1 + 0.5 + 0.25 + 0.0625 = 1.8125.

To convert a negative fractional number, first add two to the number and then use the same algorithm as already shown.

16-bit fractional numbers use a format similar to that of 8-bit fractional numbers; the high eight bits have the same format as the 8-bit format. The low eight bits are only an increase of accuracy of the 8-bit format; while the 8-bit format has an accuracy of $\pm 2^{-8}$, the16-bit format has an accuracy of $\pm 2^{-16}$. Then again, the 32-bit fractional numbers are an increase of accuracy to the 16-bit fractional numbers. Note the important difference between integers and fractional numbers when extra byte(s) are used to store the number: while the accuracy of the numbers is increased when fractional numbers are used, the range of numbers that may be represented is extended when integers are used.

As mentioned earlier, using signed fractional numbers in the range [-1, 1> has one main advantage to integers: when multiplying two numbers in the range [-1, 1>, the result will be in the range [-1, 1], and an approximation (the highest byte(s)) of the result may be stored in the same number of bytes as the factors, with one exception: when both factors are -1, the product should be 1, but since the number 1 cannot be represented using this number format, the FMULS instruction will instead place the number -1 in R1:R0. The user should therefore assure that at least one of the operands is not -1 when using the FMULS instruction. The 16-bit x 16-bit fractional multiply also has this restriction.

4.1. Example 5 – Basic Usage 8-bit x 8-bit = 16-bit Signed Fractional Multiply

This example shows an assembly code that reads the port B input value and multiplies this value with a fractional constant (-0.625) before storing the result in register pair R17:R16.

```
in r16,PINB ; Read pin values
ldi r17,$B0 ; Load -0.625 into r17
fmuls r16,r17 ; r1:r0 = r17 * r16
movw r17:r16,r1:r0 ; Move the result to the r17:r16 register pair
```

Note that the usage of the FMULS (and FMUL) instructions is very similar to the usage of the MULS and MUL instructions.

4.2. Example 6 – Multiply-accumulate Operation

The example below uses data from the ADC. The ADC should be configured so that the format of the ADC result is compatible with the fractional two's complement format. For the ATmega83/163, this means that the ADLAR bit in the ADMUX I/O register is set and a differential channel is used. (The ADC result is normalized to one.)

```
ldi r23,$62 ; Load highbyte of fraction 0.771484375
ldi r22,$C0 ; Load lowbyte of fraction 0.771484375
in r20,ADCL ; Get lowbyte of ADC conversion
in r21,ADCH ; Get highbyte of ADC conversion
call fmac16x16_32 ;Call routine for signed fractional multiply accumulate
```

The registers R19:R18:R17:R16 will be incremented with the result of the multiplication of 0.771484375 with the ADC conversion result. In this example, the ADC result is treated as a signed fraction number. We could also treat it as a signed integer and call it "mac16x16_32" instead of "fmac16x16_32". In this case, the 0.771484375 should be replaced with an integer.

5. Comment on Implementations

All 16-bit x 16-bit = 32-bit functions implemented here start by clearing the R2 register, which is just used as a “dummy” register with the “add with carry” (ADC) and “subtract with carry” (SBC) operations. These operations do not alter the contents of the R2 register. If the R2 register is not used elsewhere in the code, it is not necessary to clear the R2 register each time these functions are called, but only once prior to the first call to one of the functions.

6. Revision History

Doc. Rev.	Date	Comments
1631D	10/2016	New template
1631C	06/2002	-
1631B	-	-
1631A	-	Initial document release

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**

© 2016 Atmel Corporation. / Rev.: Atmel-1631D-Using-the-AVR-Hardware-Multiplier_AVR201_Application Note-10/2016

Atmel®, Atmel logo and combinations thereof, Enabling Unlimited Possibilities®, AVR®, megaAVR®, and others are registered trademarks or trademarks of Atmel Corporation in U.S. and other countries. Other terms and product names may be trademarks of others.

DISCLAIMER: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER: Atmel products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death (“Safety-Critical Applications”) without an Atmel officer's specific written consent. Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Atmel products are not designed nor intended for use in military or aerospace applications or environments unless specifically designated by Atmel as military-grade. Atmel products are not designed nor intended for use in automotive applications unless specifically designated by Atmel as automotive-grade.