MSP430 Competitive Benchmarking

Greg Morton *MSP430*

ABSTRACT

This application report contains the results from benchmarking the MSP430 against microcontrollers from other vendors. IAR Systems' Embedded Workbench™ development platform was used to build and execute—in simulation mode—a set of simple applications. Various applications were executed on each microcontroller to benchmark different aspects of the microcontroller's performance. For each application built, the code size and the number of instruction cycles required for execution were recorded. Figure 1 and Figure 2 show the total code size and total instruction cycle count respectively for all applications built and executed on each microcontroller.

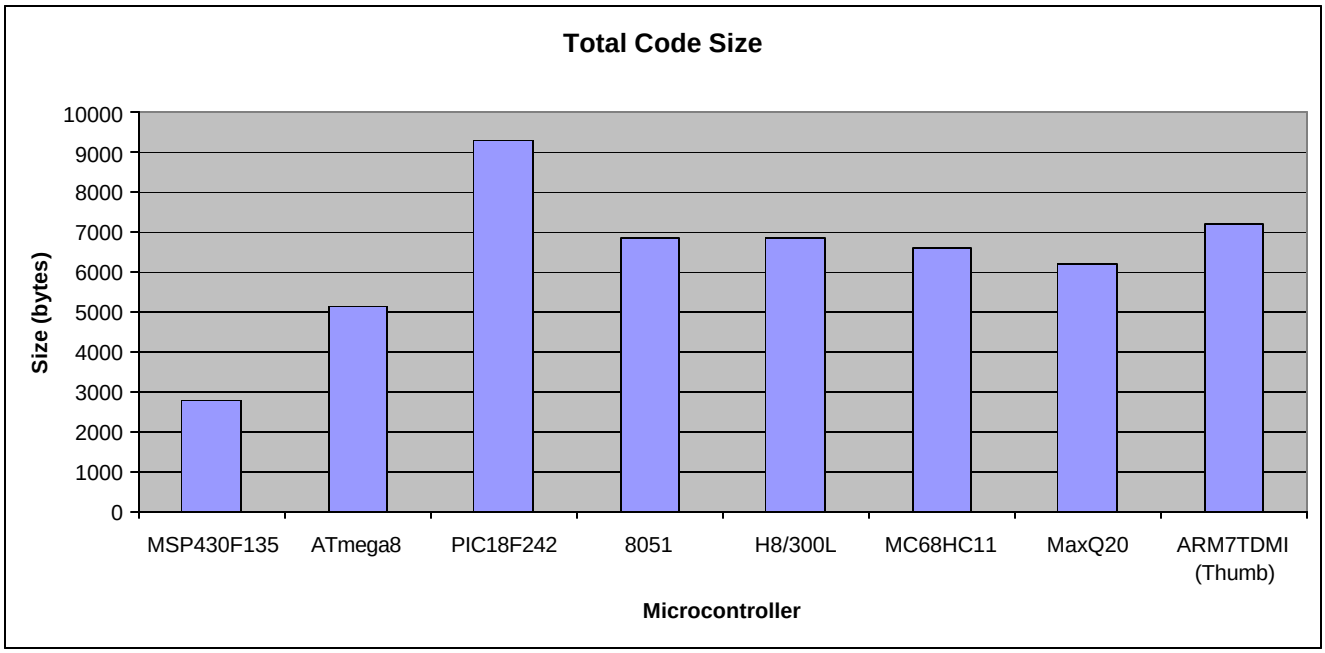

Figure 1. Total Code Size

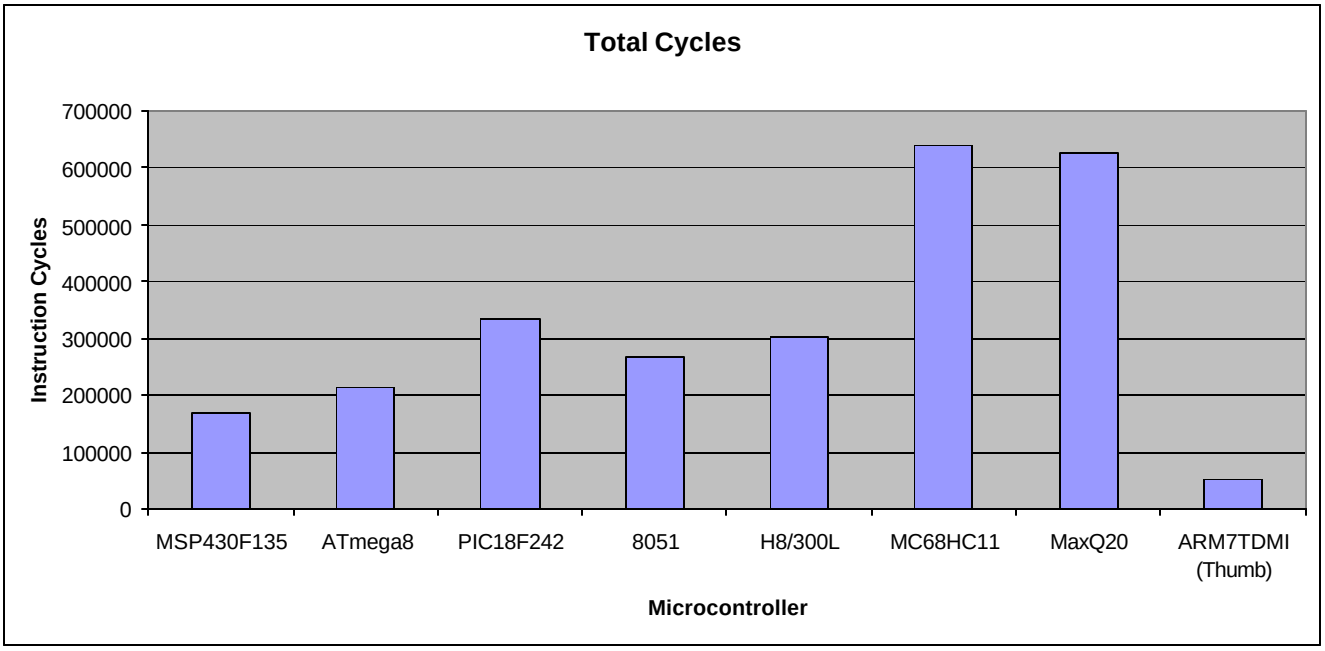

Figure 2. Total Instruction Cycle Counts

Table 1 displays total code size and total instruction counts for each microcontroller normalized against the MSP430.

Table 1. Normalized Results

Microcontroller	Total Code Size	Total Instruction Cycle Count
MSP430F135	1.00	1.00
ATmega8	1.82	1.26
PIC18F242	3.31	1.98
8051	2.45	1.58
H8/300L	2.45	1.78
MC68HC11	2.36	3.76
MaxQ20	2.20	3.68
ARM7TDMI	2.57	0.31

Table 2 contains code sizes (in bytes) for the applications benchmarked on each of the microcontrollers.

Table 2. Code Size (Bytes)

Application	MSP430F135	ATmega8	PIC18F242	8051	H8/300L	MC68HC11	MaxQ20	ARM7
8-bit Math	172	116	386	141	354	285	352	660
8-bit Matrix	118	364	676	615	356	380	378	408
8-bit Switch	180	342	404	209	362	387	202	504
16-bit Math	172	174	598	361	564	315	286	676
16-bit Matrix	156	570	846	825	450	490	526	428
16-bit Switch	178	388	572	326	404	405	188	504
32-bit Math	250	316	960	723	876	962	338	620
Floating-point Math	662	1042	1778	1420	1450	1429	1596	1556
FIR Filter	668	1292	2146	1915	1588	1470	1828	1420
Matrix Multiplication	252	510	936	345	462	499	494	432
Total	2808	5114	9302	6880	6866	6622	6188	7208

Table 3 contains instruction cycle counts for the applications benchmarked on each of the microcontrollers.

Table 3. Instruction Cycle Counts

Application	MSP430F135	ATmega8	PIC18F242	8051	H8/300L	MC68HC11	MaxQ20	ARM7
8-bit Math	299	157	318	112	680	387	421	185
8-bit Matrix	2899	5300	20045	17744	9098	15412	31691	2227
8-bit Switch	50	131	109	84	388	214	58	146
16-bit Math	343	319	625	426	802	508	815	259
16-bit Matrix	5784	24426	27021	29468	15280	23164	60214	2998
16-bit Switch	49	144	163	120	398	230	51	146
32-bit Math	792	782	1818	2937	1756	1446	1034	115
Floating-point Math	1207	1601	1599	2487	2458	4664	1943	108
FIR Filter	152193	164793	248655	206806	245588	567139	464558	43191
Matrix Multiplication	6633	16027	36190	9454	26750	26874	66534	2918
Total	170249	213680	336543	269638	303198	640038	627319	52293

Impact of Hardware Multiplier and Full Compiler Optimization

To show the impact of the MSP430's hardware multiplier along with full compiler optimization, the MSP430F135 was benchmarked against the MSP430F149. The MSP430F149 contains a hardware multiplier whereas the MSP430F135 does not. IAR's latest C compiler for the MSP430, version 3.10A, was used to build and execute the benchmarking applications for this comparison. Table 4 shows the results.

Table 4. Impact of Hardware Multiplier and Compiler Optimization

Application	MSP430F135 Optimization: None Hardware Multiplier: No		MSP430F149 Optimization: None Hardware Multiplier: Yes		MSP430F149 Optimization: Full Hardware Multiplier: Yes	
	Bytes	Cycles	Bytes	Cycles	Bytes	Cycles
8 Bit Math	172	299	156	268	136	249
8 Bit 2 Dim Matrix	118	2899	118	2899	124	1110
8 Bit Switch Case	180	50	180	50	178	49
16 Bit Math	172	343	156	268	134	248
16 Bit 2 Dim Matrix	156	5784	156	5784	112	1174
16 Bit Switch Case	178	49	178	49	176	48
32 Bit Math	250	792	248	607	222	583
Floating-point Math	662	1207	702	1012	678	993
FIR Filter	688	152193	728	143694	732	137161
Matrix Multiplication	252	6633	236	5399	166	2847
Total	2828	170249	2858	160030	2658	144462

Appendix

Benchmarking Applications

In order to benchmark various aspects of a microcontroller's performance, the following set of simple applications was executed—in simulation mode—for each microcontroller. Source code can be found in the appendix.

8-bit_math.c – source file containing three math functions. One function performs addition of two 8-bit numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

16-bit_math.c – source file containing three math functions. One function performs addition of two 16-bit numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

32-bit_math.c – source file containing three math functions. One function performs addition of two 32-bit numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

floating_point_math.c – source file containing three math functions. One function performs addition of two floating-point numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

8-bit_switch_case.c – source file with one function containing a switch statement having 16 cases. An 8-bit value is used to select a particular case. The "main()" function calls the "switch" function with an input parameter selecting the last case.

16-bit_switch_case.c – source file with one function containing a switch statement having 16 cases. A 16-bit value is used to select a particular case. The "main()" function calls the "switch" function with an input parameter selecting the last case.

8-bit_2-dim_matrix.c – source file containing 3 two-dimensional arrays containing 8-bit values—one of which is initialized. The "main()" function copies values from array 1 to array 2, then from array 2 to array 3.

16-bit_2-dim_matrix.c – source file containing 3 two-dimensional arrays containing 16-bit values—one of which is initialized. The "main()" function copies values from array 1 to array 2, then from array 2 to array 3.

fir_filter.c – source file containing code that calculates the output from a 17-coefficient tap filter using simulated ADC input data.

matrix_multiplication.c – source file containing code which multiplies a 3x4 matrix by a 4x5 matrix.

Compiler

The “C” compiler bundled with IAR Systems’ Embedded Workbench™ Integrated Development Environment (IDE) was used to build the benchmarking applications. Evaluation copies of the IDE were obtained for each microcontroller from IAR Systems’ web site located at http://www.iar.com. Table 5 lists the “C” compiler version used to build the benchmarking applications for each microcontroller. Code size was recorded for each application built.

All applications were built with compiler optimization set to “none”. This was done to minimize the compiler’s impact on the results. For example, with optimization enabled, the compiler for the 8051 was able to calculate results for some math functions at compile-time. The optimization algorithm was sophisticated enough to determine that the input values to the math functions did not change. This allowed the compiler to evaluate the math function and replace the function’s code with its result.

All benchmarking applications were executed in simulation mode. The number of instructions required to execute each application were recorded.

Table 5. “C” Compiler Versions

Microcontroller	IAR C Compiler Version
MSP430F135	2.21B
Atmel ATmega8	3.10C
Microchip PIC18F242	6.10A
Generic 8051	2.12A
Renesas H8/300L	4.20A
Motorola MC68HC11	4.45A
MaxQ20	3.0B
ARM7TDMI	4.11A

Benchmarking Application Source Code

The following are the “C” source code files for the benchmarking applications used in this document.

8-bit Math.c

```
/*****************************************************************************
*
*          Name    : 8-bit Math
*          Purpose : Benchmark 8-bit math functions.
*
*****************************************************************************/


typedef unsigned char UInt8;

UInt8 add(UInt8 a, UInt8 b)
{
  return (a + b);
}

UInt8 mul(UInt8 a, UInt8 b)
{
  return (a * b);
}

UInt8 div(UInt8 a, UInt8 b)
{
  return (a / b);
}

void main(void)
{
  volatile UInt8 result[4];

  result[0] = 12;
  result[1] = 3;

  result[2] = add(result[0], result[1]);

  result[1] = mul(result[0], result[2]);

  result[3] = div(result[1], result[2]);

  return;
}
```

8-bit 2-dim Matrix.c

```
/*****************************************************************************
*
*          Name    : 8-bit 2-dim Matrix
*          Purpose : Benchmark copying 8-bit values.
*
*****************************************************************************/

typedef unsigned char UInt8;

const UInt8 m1[16][4] = {
                   {0x12, 0x56, 0x90, 0x34},
                   {0x78, 0x12, 0x56, 0x90},
                   {0x34, 0x78, 0x12, 0x56},
                   {0x90, 0x34, 0x78, 0x12},

                   {0x12, 0x56, 0x90, 0x34},
                   {0x78, 0x12, 0x56, 0x90},
                   {0x34, 0x78, 0x12, 0x56},
                   {0x90, 0x34, 0x78, 0x12},
                   {0x12, 0x56, 0x90, 0x34},
                   {0x78, 0x12, 0x56, 0x90},
                   {0x34, 0x78, 0x12, 0x56},
                   {0x90, 0x34, 0x78, 0x12},
                   {0x12, 0x56, 0x90, 0x34},
                   {0x78, 0x12, 0x56, 0x90},
                   {0x34, 0x78, 0x12, 0x56},
                   {0x90, 0x34, 0x78, 0x12}
                 };


void main (void)
{
  int i, j;

  volatile UInt8 m2[16][4], m3[16][4];

  for(i = 0; i < 16; i++)
  {
    for(j=0; j < 4; j++)
    {
      m2[i][j] = m1[i][j];
      m3[i][j] = m2[i][j];
    }
  }

  return;
}
```

8-bit Switch Case.c

```
/*******************************************************************************
*
*          Name    : 8-bit Switch Case
*          Purpose : Benchmark accessing switch statement using 8-bit value.
*
*******************************************************************************/

typedef unsigned char UInt8;

UInt8 switch_case(UInt8 a)
{
   UInt8 output;

   switch (a)
   {
      case 0x01:
      output = 0x01;
      break;

      case 0x02:
      output = 0x02;
      break;

      case 0x03:
      output = 0x03;
      break;

      case 0x04:
      output = 0x04;
      break;

      case 0x05:
      output = 0x05;
      break;

      case 0x06:
      output = 0x06;
      break;

      case 0x07:
      output = 0x07;
      break;

      case 0x08:
      output = 0x08;
      break;

      case 0x09:
      output = 0x09;
      break;
```

```
      case 0x0a:
      output = 0x0a;
      break;

      case 0x0b:
      output = 0x0b;
      break;

      case 0x0c:
      output = 0x0c;
      break;

      case 0x0d:
      output = 0x0d;
      break;

      case 0x0e:
      output = 0x0e;
      break;

      case 0x0f:
      output = 0x0f;
      break;

      case 0x10:
      output = 0x10;
      break;
      } /* end switch*/

  return (output);
}

void main(void)
{
  volatile UInt8 result;

  result = switch_case(0x10);

  return;
}
```

16-bit Math.c

```
/*****************************************************************************
*
*          Name    : 16-bit Math
*          Purpose : Benchmark 16-bit math functions.
*
*****************************************************************************/

typedef unsigned short UInt16;


UInt16 add(UInt16 a, UInt16 b)
{
  return (a + b);
}

UInt16 mul(UInt16 a, UInt16 b)
{
  return (a * b);
}

UInt16 div(UInt16 a, UInt16 b)
{
  return (a / b);
}

void main(void)
{
  volatile UInt16 result[4];

  result[0] = 231;
  result[1] = 12;

  result[2] = add(result[0], result[1]);

  result[1] = mul(result[0], result[2]);

  result[3] = div(result[1], result[2]);

  return;
}
```

16-bit 2-dim Matrix.c

```
/*******************************************************************************
*
*           Name    : 16-bit 2-dim Matrix
*           Purpose : Benchmark copying 16-bit values.
*
*******************************************************************************/

typedef unsigned short UInt16;

const UInt16 m1[16][4] = {
                    {0x1234, 0x5678, 0x9012, 0x3456},
                    {0x7890, 0x1234, 0x5678, 0x9012},
                    {0x3456, 0x7890, 0x1234, 0x5678},
                    {0x9012, 0x3456, 0x7890, 0x1234},
                    {0x1234, 0x5678, 0x9012, 0x3456},
                    {0x7890, 0x1234, 0x5678, 0x9012},
                    {0x3456, 0x7890, 0x1234, 0x5678},
                    {0x9012, 0x3456, 0x7890, 0x1234},
                    {0x1234, 0x5678, 0x9012, 0x3456},
                    {0x7890, 0x1234, 0x5678, 0x9012},
                    {0x3456, 0x7890, 0x1234, 0x5678},
                    {0x9012, 0x3456, 0x7890, 0x1234},
                    {0x1234, 0x5678, 0x9012, 0x3456},
                    {0x7890, 0x1234, 0x5678, 0x9012},
                    {0x3456, 0x7890, 0x1234, 0x5678},
                    {0x9012, 0x3456, 0x7890, 0x1234}
                  };


void main(void)
{
  int i, j;

  volatile UInt16 m2[16][4], m3[16][4];

  for(i = 0; i < 16; i++)
  {
    for(j = 0; j < 4; j++)
    {
      m2[i][j] = m1[i][j];
      m3[i][j] = m2[i][j];
    }
  }

  return;
}
```

16-bit Switch Case.c

```
/*******************************************************************************
*
*          Name    : 16-bit Switch Case
*          Purpose : Benchmark accessing switch statement using 16-bit value.
*
*******************************************************************************/

typedef unsigned short UInt16;

UInt16 switch_case(UInt16 a)
{
   UInt16 output;

   switch (a)
   {
      case 0x0001:
      output = 0x0001;
      break;

      case 0x0002:
      output = 0x0002;
      break;

      case 0x0003:
      output = 0x0003;
      break;

      case 0x0004:
      output = 0x0004;
      break;

      case 0x0005:
      output = 0x0005;
      break;

      case 0x0006:
      output = 0x0006;
      break;

      case 0x0007:
      output = 0x0007;
      break;

      case 0x0008:
      output = 0x0008;
      break;

      case 0x0009:
      output = 0x0009;
      break;
```

```
      case 0x000a:
      output = 0x000a;
      break;

      case 0x000b:
      output = 0x000b;
      break;

      case 0x000c:
      output = 0x000c;
      break;

      case 0x000d:
      output = 0x000d;
      break;

      case 0x000e:
      output = 0x000e;
      break;

      case 0x000f:
      output = 0x000f;
      break;

      case 0x0010:
      output = 0x0010;
      break;
      } /* end switch*/

  return (output);
}


void main(void)
{
  volatile UInt16 result;

  result = switch_case(0x0010);

  return;
}
```

32-bit Math.c

```
/*****************************************************************************
*
*          Name    : 32-bit Math
*          Purpose : Benchmark 32-bit math functions.
*
*****************************************************************************/

#include <math.h>

typedef unsigned long UInt32;

UInt32 add(UInt32 a, UInt32 b)
{
  return (a + b);
}

UInt32 mul(UInt32 a, UInt32 b)
{
  return (a * b);
}

UInt32 div(UInt32 a, UInt32 b)
{
  return (a / b);
}

void main(void)
{
  volatile UInt32 result[4];

  result[0] = 43125;
  result[1] = 14567;

  result[2] = add(result[0], result[1]);

  result[1] = mul(result[0], result[2]);

  result[3] = div(result[1], result[2]);

  return;
}
```

Floating-point Math.c

```
/*****************************************************************************
*
*           Name    : Floating-point Math
*           Purpose : Benchmark floating-point math functions.
*
*****************************************************************************/

float add(float a, float b)
{
  return (a + b);
}

float mul(float a, float b)
{
  return (a * b);
}

float div(float a, float b)
{
  return (a / b);
}

void main(void)
{
  volatile float result[4];

  result[0] = 54.567;
  result[1] = 14346.67;

  result[2] = add(result[0], result[1]);

  result[1] = mul(result[0], result[2]);

  result[3] = div(result[1], result[2]);

  return;
}
```

FIR Filter.c

```
/*****************************************************************************
*
*          Name    : FIR Filter
*          Purpose : Benchmark a FIR filter. The input values for the filter
*                    is an array of 51 16-bit values. The order of the filter
*                    17.
*
*****************************************************************************/

#ifdef MSP430
#include "msp430x13x.h"
#endif

#include <math.h>

#define FIR_LENGTH 17

const float COEFF[FIR_LENGTH] =
{
   -0.000091552734,
    0.000305175781,
    0.004608154297,
    0.003356933594,
   -0.025939941406,
   -0.044006347656,
    0.063079833984,
    0.290313720703,
    0.416748046875,
    0.290313720703,
    0.063079833984,
   -0.044006347656,
   -0.025939941406,
    0.003356933594,
    0.004608154297,
    0.000305175781,
   -0.000091552734
};



/* The following array simulates input A/D converted values */

const unsigned int INPUT[] =
{
    0x0400, 0x0800, 0x0C00, 0x1000, 0x1400, 0x1800, 0x1C00, 0x2000,
    0x2400, 0x2000, 0x1C00, 0x1800, 0x1400, 0x1000, 0x0C00, 0x0800,
    0x0400, 0x0400, 0x0800, 0x0C00, 0x1000, 0x1400, 0x1800, 0x1C00,
    0x2000, 0x2400, 0x2000, 0x1C00, 0x1800, 0x1400, 0x1000, 0x0C00,
    0x0800, 0x0400, 0x0400, 0x0800, 0x0C00, 0x1000, 0x1400, 0x1800,
    0x1C00, 0x2000, 0x2400, 0x2000, 0x1C00, 0x1800, 0x1400, 0x1000,
    0x0C00, 0x0800, 0x0400
```

```
};


void main(void)
{
  int i, y; /* Loop counters */

  volatile float OUTPUT[36];

#ifdef MSP430
  WDTCTL = WDTPW + WDTHOLD; /* Stop watchdog timer */
#endif

  for(y = 0; y < 36; y++)
  {
    for(i = 0; i < FIR_LENGTH/2; i++)
    {
      OUTPUT[y] = COEFF[i] * ( INPUT[y + 16 - i] + INPUT[y + i] );
    };

    OUTPUT[y] = OUTPUT[y] + ( INPUT[y + 16 - i] * COEFF[i] );
  }

  return;
}
```

Matrix Multiplication.c

```
/*****************************************************************************
*
*          Name    : Matrix Multiplication
*          Purpose : Benchmark multiplying a 3x4 matrix by a 4x5 matrix.
*                    Matrix contains 16-bit values.

*
*****************************************************************************/

typedef unsigned short UInt16;

const UInt16 m1[3][4] = {
  {0x01, 0x02, 0x03, 0x04},
  {0x05, 0x06, 0x07, 0x08},
  {0x09, 0x0A, 0x0B, 0x0C}
};

const UInt16 m2[4][5] = {
  {0x01, 0x02, 0x03, 0x04, 0x05},
  {0x06, 0x07, 0x08, 0x09, 0x0A},
  {0x0B, 0x0C, 0x0D, 0x0E, 0x0F},
  {0x10, 0x11, 0x12, 0x13, 0x14}
};

void main(void)
{
  int m, n, p;

  volatile UInt16 m3[3][5];

  for(m = 0; m < 3; m++)
  {
    for(p = 0; p < 5; p++)
    {
      m3[m][p] = 0;

      for(n = 0; n < 4; n++)
      {
        m3[m][p] += m1[m][n] * m2[n][p];
      }
    }
  }

  return;
}
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265