Freescale Semiconductor
Application Note

Document Number: AN3884
Rev. 0, 07/2009

Efficient Low-Level Software Development for the i.MX Platform

by *Multimedia Applications Division*
Freescale Semiconductor, Inc.
Austin, TX

This application note describes how to write efficient C code for the i.MX platform. Because the i.MX platform is based on the ARM® cores, this application note is applicable for all i.MX devices. This document uses standard C for the examples; however, the techniques described apply equally to C++. The examples in this document are simple and concise in order to highlight specific issues.

Currently, the ARM family accounts for approximately 75% of all embedded 32-bit RISC CPUs, making it one of the most widely used 32-bit architectures. ARM CPUs are found in most areas of consumer electronics from portable devices (PDAs, mobile phones, media players, handheld gaming units, and calculators) to computer peripherals (hard drives, desktop, routers). This application note assumes the reader has some knowledge of ARM assembly programming.

Contents

1 C Compiler Overview

Since C language is the most common programming language for embedded systems, this document assumes the reader is familiar with interpreting C code. To write efficient C code, programmers should be aware of the following issues:

- Areas where the C compiler has to be conservative
- The limits of the processor architecture the C compiler is mapping to
- The limits of a specific C compiler vendor.

The examples in this document have been tested using armcc from the ARM developer suite using the following command line:

```
armcc --c90 --cpu=ARM926EJ-S -O0
```

which means that the armcc compiler for ISO standard C (1990) source language is used with no space or time optimizations. The processor target is the ARM926EJ-S™ with little endian byte order.

2 Basic C Data Types

The ARM processors have 32-bit registers and a 32-bit Arithmetic Logic Unit (ALU). The ARM architecture is a RISC load/store architecture which means that values are loaded from memory into registers before they are used. The armcc compiler uses the data type mappings shown in Table 1. For ARM microprocessors, the char type is unsigned for ARM compilers which can cause problems when porting code from other processor architectures.

Table 1. Data Type Mapping

C Data Type	Implementation
char	Unsigned 8-bit char
short	Signed 16-bit (halfword)
int	Signed 32-bit word
long	Signed 32-bit word
long long	Signed 64-bit double word

2.1 Local Variable Types

Most ARM data processing operations are only 32-bit. Thus, it is recommended to use a 32-bit data type, int or long, for local variables wherever possible. To see the effect of local variable types, consider Example 1.

Example 1. Char Type Local Variable

```
void LocalVariableV1(void)
{

    char i;

    for(i=0;i<0xff;i++)
    {
      // DO SOMETHING
    }


}
```

At first, it looks as if declaring `i` as a char is efficient and that char uses less register space or less space on the ARM stack than int. Unfortunately, both of these assumptions are incorrect since all ARM registers are 32-bit and all stack entries are at least 32-bit. In addition, to implement `i++`, the compiler must account for the case when `i` is equal to 0xFF. The compiler output for this function is shown in Example 2.

Example 2. Char Type Local Variable Compiler Output

```
LocalVariableV1
$a
.text
    0x00000000:    e3a00000    ....    MOV     r0,#0
    0x00000004:    ea000001    ....    B       {pc} + 0xc   ; 0x10
    0x00000008:    e2801001    ....    ADD     r1,r0,#1
    0x0000000c:    e20100ff    ....    AND     r0,r1,#0xff
    0x00000010:    e35000ff    ..P.    CMP     r0,#0xff
    0x00000014:    bafffffb    ....    BLT     {pc} - 0xc   ; 0x8
    0x00000018:    e12fff1e    ../.    BX      r14
```

The compiler increments `i` by 1 and inserts an **and** instruction to reduce `i` to the range 0 to 255 before the comparison with 0xFF. Example 3 shows what happens when `i` is declared as short type.

Example 3. Short Type Local Variable

```
void LocalVariableV2(void)
{

    short i;

    for(i=0;i<0xff;i++)
    {
      // DO SOMETHING
    }


}
```

The result (shown in Example 4) is a little less efficient since the compiler inserts **lsl** and **asr** instructions to reduce `i` to the range 0 to 65535 before the comparison with 0xFF.

Example 4. Short Type Local Variable Compiler Output

```
LocalVariableV2
    0x0000001c:    e3a00000    ....    MOV     r0,#0
    0x00000020:    ea000002    ....    B       {pc} + 0x10   ; 0x30
    0x00000024:    e2801001    ....    ADD     r1,r0,#1
    0x00000028:    e1a01801    ....    LSL     r1,r1,#16
    0x0000002c:    e1a00841    A...    ASR     r0,r1,#16
    0x00000030:    e35000ff    ..P.    CMP     r0,#0xff
    0x00000034:    bafffffa    ....    BLT     {pc} - 0x10   ; 0x24
    0x00000038:    e12fff1e    ../.    BX      r14
```

Example 5 shows `i` declared as int type.

Example 5. Int Type Local Variable

```
void LocalVariableV3(void)
{

    int i;

    for(i=0;i<0xff;i++)
    {
      // DO SOMETHING
    }

}
```

The routine with `i` declared as int type, shown in Example 6, is more space efficient because it requires only six assembly instructions instead of seven for char type or eight for short type.

Example 6. Int Type Local Variable Compiler Output

```
LocalVariableV3
    0x0000003c:    e3a00000    ....    MOV     r0,#0
    0x00000040:    ea000000    ....    B       {pc} + 0x8    ; 0x48
    0x00000044:    e2800001    ....    ADD     r0,r0,#1
    0x00000048:    e35000ff    ..P.    CMP     r0,#0xff
    0x0000004c:    bafffffc    ....    BLT     {pc} - 0x8    ; 0x44
    0x00000050:    e12fff1e    ../.    BX      r14
```

By using `i` as long type, the output (in Example 7) is similar to the int type version, since int and long are both signed 32-bit word for the armcc compiler.

Example 7. Long Type Local Variable Compiler Output

```
LocalVariableV4
    0x00000054:    e3a00000    ....    MOV     r0,#0
    0x00000058:    ea000000    ....    B       {pc} + 0x8    ; 0x60
    0x0000005c:    e2800001    ....    ADD     r0,r0,#1
    0x00000060:    e35000ff    ..P.    CMP     r0,#0xff
    0x00000064:    bafffffc    ....    BLT     {pc} - 0x8    ; 0x5c
    0x00000068:    e12fff1e    ../.    BX      r14
```

2.2 Function Argument Type

The previous section shows that using int or long types as local variables increases performance and reduces code size. The same is true for function arguments. Example 8 shows a function using char type.

Example 8. Char Type Function Argument

```
char FunctionArgumentV1(char a,char b)
{
   return a + b;
}
```

The input values a and b are passed in 32-bit ARM registers. The return value is also passed in a 32-bit register. For the armcc compiler, function arguments are passed narrow and values are returned narrow. That means the caller casts argument values and the callee casts return values. The armcc output for FunctionArgumentV1, shown in Example 9, demonstrates that the compiler casts the return value to char type with an **and** assembly instruction.

Example 9. Char Type Function Argument Compiler Output

```
FunctionArgumentV1
    0x000000ac:    e1a02000    . ..    MOV     r2,r0
    0x000000b0:    e0820001    ....    ADD     r0,r2,r1
    0x000000b4:    e20000ff    ....    AND     r0,r0,#0xff
    0x000000b8:    e12fff1e    ../.    BX      r14
```

Example 10 and Example 11 show the caller for FunctionArgumentV1.

Example 10. Function Argument Caller

```
void main (void)
{
    FunctionArgumentV1(0xabcdef,0xabcdef);
}
```

The caller uses short parameters when the arguments are char type. The callee casts the input values since the caller has implicitly ensured that the arguments are in the range of char type.

Example 11. Function Argument Caller Compiler Output

```
main
    0x00000484:    e52de004    ..-.    PUSH    {r14}
    0x00000488:    e3a010ef    ....    MOV     r1,#0xef
    0x0000048c:    e3a000ef    ....    MOV     r0,#0xef
    0x00000490:    ebfffffe    ....    BL      FunctionArgumentV1
    0x00000494:    e49df004    ....    POP     {pc}
```

Example 12 shows the function arguments modified to short type.

Example 12. Short Type Function Argument

```
short FunctionArgumentV2(short a,short b)
{
   return a + b;
}
```

The compiler output (in Example 13) shows that the compiler inserts **lsl** and **asr** instructions to cast `i` to the range of short type before returning the value through the r0 register.

Example 13. Short Type Function Argument Compiler Output

```
FunctionArgumentV2
    0x000000bc:    e1a02000    . . .    MOV      r2,r0
    0x000000c0:    e0820001    . . .    ADD      r0,r2,r1
    0x000000c4:    e1a00800    . . .    LSL      r0,r0,#16
    0x000000c8:    e1a00840    @ . .    ASR      r0,r0,#16
    0x000000cc:    e12fff1e    . ./.    BX       r14
```

Finally, Example 14 uses int type as the function arguments.

Example 14. Int Type Function Argument

```
int FunctionArgumentV3(int a,int b)
{
   return a + b;
}
```

The compiler output (in Example 15) shows that the arguments are passed to the function through the r0 and r1 registers. There is no casting for the return value since the resulting sum matches the return type. FunctionArgumentV3 only requires three instructions.

Example 15. Int Type Function Argument Compiler Output

```
FunctionArgumentV3
    0x000000d0:    e1a02000    . . .    MOV      r2,r0
    0x000000d4:    e0820001    . . .    ADD      r0,r2,r1
    0x000000d8:    e12fff1e    . ./.    BX       r14
```

Undoubtedly, char or short type function arguments and return values introduce extra casts. These extra casts increase code size and reduce performance. It is more efficient to use int or long types for function arguments and return values, even if only an 8-bit value is being passed.

3 C Loops

This section describes the most efficient ways to code for and while loops on the ARM architecture and the specific implementations for the armcc compiler.

3.1 For Loop

Example 16 shows a for loop implemented on the ARM platform. This example uses the ForLoopV1 function which has a fixed number of iterations.

Example 16. For Loop

```
void ForLoopV1(void)
{
  unsigned int i;

  for(i=0;i<0x1fff;i++)
  {
    // DO SOMETHING
  }

}
```

The compiler output is shown in Example 17. The first line sets `i` to 0. Then, the output jumps to address 0x110 where two **subs**/**subcs** instructions compare if `i` is less than 0x1FFFF. If the condition is true, the code jumps to address 0x10C where `i` is incremented by 1. If the condition is not true, the code returns from the function.

Example 17. For Loop Compiler Output

```
ForLoopV1
    0x00000104:    e3a00000    ....    MOV     r0,#0
    0x00000108:    ea000000    ....    B       {pc} + 0x8   ; 0x110
    0x0000010c:    e2800001    ....    ADD     r0,r0,#1
    0x00000110:    e250cc1f    ..P.    SUBS    r12,r0,#0x1f00
    0x00000114:    225cc0ff    ..\"    SUBSCS  r12,r12,#0xff
    0x00000118:    3afffffb    ...:    BCC     {pc} - 0xc   ; 0x10c
    0x0000011c:    e12fff1e    ../.    BX      r14
```

This code can be improved using a loop that counts down to zero and uses the continuation condition `i` != 0 as shown in Example 18.

Example 18. Improved For Loop

```
void ForLoopV2(void)
{
  unsigned int i;

  for(i=0x1fff;i!=0;i--)
  {
    // DO SOMETHING
  }

}
```

The new compiler output (in Example 19) does not allocate memory or use a register to store the termination value. In addition, the comparison with zero only requires one instruction.

Example 19. Improved For Loop Compiler Output

```
ForLoopV2
    0x00000120:    e59f015c    \...    LDR     r0,[pc,#348]  ; [0x284] = 0x1fff
    0x00000124:    ea000000    ....    B       {pc} + 0x8   ; 0x12c
    0x00000128:    e2400001    ..@.    SUB     r0,r0,#1
    0x0000012c:    e3500000    ..P.    CMP     r0,#0
    0x00000130:    1afffffc    ....    BNE     {pc} - 0x8   ; 0x128
    0x00000134:    e12fff1e    ../.    BX      r14
```

The LoopV2 function loads the iteration variable from memory address 0x1FFFF into the r0 register. Then, it jumps to address 0x12C where compares the iteration variable with zero. If the iteration variable is equal to zero, the code returns from the function. If the iteration variable is not zero, jumps to address 0x128 and decrements the iteration variable by 1. Using unsigned int iteration variables is more efficient because it does not require additional casting.

3.2 Do-While Loop

The optimizations used in for loops can also be used for do-while loops. Example 20 shows a simple do-while loop function called DoWhileLoopV1.

Example 20. Do-While Loop

```
void DoWhileLoopV1(void)
{
   unsigned int n=0;

   do
   {
     // DO SOMETHING

   }while(++n<255);
}
```

For the do-while loop that uses an iteration variable that is incremented by 1, the compiler output requires seven instructions as shown in Example 21.

Example 21. Do-While Loop Compiler Output

```
DoWhileLoopV1
    0x00000160:    e3a00000    ....    MOV     r0,#0
    0x00000164:    e1a00000    ....    MOV     r0,r0
    0x00000168:    e2801001    ....    ADD     r1,r0,#1
    0x0000016c:    e1a00001    ....    MOV     r0,r1
    0x00000170:    e35100ff    ..Q.    CMP     r1,#0xff
    0x00000174:    3afffffb    ...:    BCC     {pc} - 0xc   ; 0x168
    0x00000178:    e12fff1e    ../.    BX      r14
```

Example 22 decrements the iteration variable by 1 and uses the continuation condition n != 0.

Example 22. Improved Do-While Loop

```
void DoWhileLoopV2(void)
{
   unsigned int n=255;

   do
   {
     // DO SOMETHING

   }while(--n!=0);

}
```

The first instruction in Example 23 moves the immediate value 0xFF to the r0 register. Then, the code subtracts 1 from the r0 register and stores the result in the r1 register. The code moves r1 to r0 and sets the condition flags. If r0 is not equal to zero, the code jumps to address 0x184. If r0 is equal to zero, the code returns from the function. The improved do-while loop requires one less instruction than the original version since it does not store the compared value in memory.

Example 23. Improved Do-While Loop Compiler Output

```
DoWhileLoopV2
    0x0000017c:    e3a000ff    ....    MOV     r0,#0xff
    0x00000180:    e1a00000    ....    MOV     r0,r0
    0x00000184:    e2401001    ..@.    SUB     r1,r0,#1
    0x00000188:    e1b00001    ....    MOVS    r0,r1
    0x0000018c:    1afffffc    ....    BNE     {pc} - 0x8   ; 0x184
    0x00000190:    e12fff1e    ../.    BX      r14
```

4 Arrays

Fundamentally an array is simply an extension of the basic model of computer memory: an array of bytes accessible through indexes. Thus an array of a data type D is a data structure where each array item is a data container of the same data type D and can be accessed through its index. Access to multi-dimensional array items is performed according to the row-major access formula. This formula transforms, for example, a reference x[i][j] to an indirection expression *(x + (i × n) + j), where n is the row size of x.

Example 24 shows an example of a two-dimensional array implementation.

Example 24. Two-Dimensional Array

```
void ArrayV1(void)
{
    int x[100][100];

    int i=0;
    int a=0;

    for(i=0;i<10;i++)
    {
       x[i][i]=a++;
    }

}
```

Because the array is two-dimensional, according to row-major access formula, two **add** and one **mul** assembly instructions are required to calculate the index address to be accessed as shown in Example 25.

Example 25. Two-Dimensional Array Compiler Output

```
ArrayV1
$a
    0x00000274:    e24ddc9d    ..M.    SUB     r13,r13,#0x9d00
    0x00000278:    e3a00000    ....    MOV     r0,#0
    0x0000027c:    e3a01000    ....    MOV     r1,#0
    0x00000280:    e1a00000    ....    MOV     r0,r0
    0x00000284:    ea000006    ....    B       {pc} + 0x20   ; 0x2a4
    0x00000288:    e3a02019    . ..    MOV     r2,#0x19
    0x0000028c:    e0020290    ....    MUL     r2,r0,r2
    0x00000290:    e28d30c0    .0..    ADD     r3,r13,#0xc0
    0x00000294:    e0832202    ."..    ADD     r2,r3,r2,LSL #4
    0x00000298:    e7821100    ....    STR     r1,[r2,r0,LSL #2]
    0x0000029c:    e2811001    ....    ADD     r1,r1,#1
    0x000002a0:    e2800001    ....    ADD     r0,r0,#1
    0x000002a4:    e350000a    ..P.    CMP     r0,#0xa
    0x000002a8:    bafffff6    ....    BLT     {pc} - 0x20   ; 0x288
    0x000002ac:    e28ddc9d    ....    ADD     r13,r13,#0x9d00
    0x000002b0:    e12fff1e    ../.    BX      r14
```

The overload becomes more significant when using three-dimensional arrays as in Example 26.

Example 26. Three-Dimensional Array

```
void ArrayV2(void)
{
    int x[10][20][50];

    int i=0;
    int a=0;

    for(i=0;i<10;i++)
    {
       x[i][i][i]=a++;
    }

}
```

Example 27 shows that the compiler calculates the index address according to *(x + i × 100 + i × 50 + i), which requires three **add** and two **mul** instructions.

Example 27. Three-Dimensional Array Compiler Output

```
ArrayV2
    0x000002b4:    e24ddc9d    ..M.    SUB     r13,r13,#0x9d00
    0x000002b8:    e3a00000    ....    MOV     r0,#0
    0x000002bc:    e3a01000    ....    MOV     r1,#0
    0x000002c0:    e1a00000    ....    MOV     r0,r0
    0x000002c4:    ea000009    ....    B       {pc} + 0x2c   ; 0x2f0
    0x000002c8:    e3a0207d    }  ..    MOV     r2,#0x7d
    0x000002cc:    e0020290    ....    MUL     r2,r0,r2
    0x000002d0:    e28d30c0    .0..    ADD     r3,r13,#0xc0
    0x000002d4:    e0832282    ."..    ADD     r2,r3,r2,LSL #5
    0x000002d8:    e3a03019    .0..    MOV     r3,#0x19
    0x000002dc:    e0030390    ....    MUL     r3,r0,r3
    0x000002e0:    e0822183    .!..    ADD     r2,r2,r3,LSL #3
    0x000002e4:    e7821100    ....    STR     r1,[r2,r0,LSL #2]
    0x000002e8:    e2811001    ....    ADD     r1,r1,#1
    0x000002ec:    e2800001    ....    ADD     r0,r0,#1
    0x000002f0:    e350000a    ..P.    CMP     r0,#0xa
    0x000002f4:    bafffff3    ....    BLT     {pc} - 0x2c   ; 0x2c8
    0x000002f8:    e28ddc9d    ....    ADD     r13,r13,#0x9d00
    0x000002fc:    e12fff1e    ../.    BX      r14
```

Example 28 shows a simplified one-dimensional array implementation of the same size as the arrays in Example 24 and Example 26.

Example 28. Simplified One-Dimensional Array

```
void ArrayV3(void)
{
    int x[10000];

    int i=0;
    int a=0;

    for(i=0;i<10;i++)
    {
        x[i]=a++;
    }

}
```

In this case, the compiler calculates the index address according to *(x + i), which requires only one **add** instruction as shown in Example 29.

Example 29. Simplified One-Dimensional Array Compiler Output

```
ArrayV3
    0x00000300:    e24ddc9d    ..M.    SUB      r13,r13,#0x9d00
    0x00000304:    e3a00000    ....    MOV      r0,#0
    0x00000308:    e3a01000    ....    MOV      r1,#0
    0x0000030c:    e1a00000    ....    MOV      r0,r0
    0x00000310:    ea000003    ....    B        {pc} + 0x14   ; 0x324
    0x00000314:    e28d20c0    . . .   ADD      r2,r13,#0xc0
    0x00000318:    e7821100    ....    STR      r1,[r2,r0,LSL #2]
    0x0000031c:    e2811001    ....    ADD      r1,r1,#1
    0x00000320:    e2800001    ....    ADD      r0,r0,#1
    0x00000324:    e350000a    ..P.    CMP      r0,#0xa
    0x00000328:    bafffff9    ....    BLT      {pc} - 0x14   ; 0x314
    0x0000032c:    e28ddc9d    ....    ADD      r13,r13,#0x9d00
    0x00000330:    e12fff1e    ../.    BX       r14
```

Example 30 shows two different examples of array indexing. The armcc compiler translates the array indexing expression x[i] into the indirection expression *(x + i).

Example 30. Array Indexing

```
void ArrayV4(void)
{
    int x[10];
    int * px = x;
    int i;

    for(i=0;i<10;i++)
    {
        x[i]=i;
    }

}
```

```
void ArrayV5(void)
{
    int x[10];
    int * px = x;
    int i;

    for(i=0;i<10;i++)
    {
        *(px+i)=i;
    }

}
```

Thus, both functions in Example 30 generate the same compiler output shown in Example 31.

Example 31. Array Indexing Compiler Output

```
     ----------    --------    .. .    ---       ---
ArrayV4
     0x00000334:    e24dd028    (.M.    SUB       r13,r13,#0x28
     0x00000338:    e1a0100d    ....    MOV       r1,r13
     0x0000033c:    e3a00000    ....    MOV       r0,#0
     0x00000340:    ea000001    ....    B         {pc} + 0xc   ; 0x34c
     0x00000344:    e78d0100    ....    STR       r0,[r13,r0,LSL #2]
     0x00000348:    e2800001    ....    ADD       r0,r0,#1
     0x0000034c:    e350000a    ..P.    CMP       r0,#0xa
     0x00000350:    bafffffb    ....    BLT       {pc} - 0xc   ; 0x344
     0x00000354:    e28dd028    (...    ADD       r13,r13,#0x28
     0x00000358:    e12fff1e    ../.    BX        r14
ArrayV5
     0x0000035c:    e24dd028    (.M.    SUB       r13,r13,#0x28
     0x00000360:    e1a0100d    ....    MOV       r1,r13
     0x00000364:    e3a00000    ....    MOV       r0,#0
     0x00000368:    ea000001    ....    B         {pc} + 0xc   ; 0x374
     0x0000036c:    e7810100    ....    STR       r0,[r1,r0,LSL #2]
     0x00000370:    e2800001    ....    ADD       r0,r0,#1
     0x00000374:    e350000a    ..P.    CMP       r0,#0xa
     0x00000378:    bafffffb    ....    BLT       {pc} - 0xc   ; 0x36c
     0x0000037c:    e28dd028    (...    ADD       r13,r13,#0x28
     0x00000380:    e12fff1e    ../.    BX        r14
```

Multi-dimensional arrays are represented the same way as one-dimensional arrays: by a pointer holding the base address of a contiguous statically allocated segment where array items are stored. The array dimension is a logical concept, not a physical one, and the compiler translates multi-dimensional access to the underlying one-dimensional array by using the row-major formula. Therefore, the fewer dimensions in an array, the more efficient the code that is generated by the compiler.

5 Register Allocation

The compiler attempts to allocate a processor register to each local variable used in a C function. When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called spilled or swapped-out variables since they are written out to memory. To ensure good register assignment, limit the internal loop of functions to at most 12 local variables. Example 32 shows a function that has 16 local variables.

Example 32. Register Allocation with 16 Local Variables

```
int RegisterAllocationV1(void)
{
    int a=10,b=11,c=12,d=13,e=14,f=15,g=16,h=17,i=17,j=19,k=20,l=21,m=22,n=23,o=24,p=25;

    return a*b*c*d*e*f*g*h*i*j*k*l*m*n*o*p;
}
```

The armcc compiler (see Example 33) inserts four **str** instructions for storing the local variables a, b, c and d on the processor stack and four **ldr** instructions for subsequently loading them into registers to be multiplied. The result of the multiplications is returned through the r0 register.

Example 33. Register Allocation with 16 Local Variables Compiler Output

```
RegisterAllocationV1
    0x000004f0:    e92d4fff    .O-.    PUSH    {r0-r11,r14}
    0x000004f4:    e3a0000a    ....    MOV     r0,#0xa
    0x000004f8:    e58d000c    ....    STR     r0,[r13,#0xc]
    0x000004fc:    e3a0000b    ....    MOV     r0,#0xb
    0x00000500:    e58d0008    ....    STR     r0,[r13,#8]
    0x00000504:    e3a0000c    ....    MOV     r0,#0xc
    0x00000508:    e58d0004    ....    STR     r0,[r13,#4]
    0x0000050c:    e3a0000d    ....    MOV     r0,#0xd
    0x00000510:    e58d0000    ....    STR     r0,[r13,#0]
    0x00000514:    e3a0100e    ....    MOV     r1,#0xe
    0x00000518:    e3a0200f    . ..    MOV     r2,#0xf
    0x0000051c:    e3a03010    .0..    MOV     r3,#0x10
    0x00000520:    e3a0c011    ....    MOV     r12,#0x11
    0x00000524:    e3a0e012    ....    MOV     r14,#0x12
    0x00000528:    e3a04013    .@..    MOV     r4,#0x13
    0x0000052c:    e3a05014    .P..    MOV     r5,#0x14
    0x00000530:    e3a06015    .`..    MOV     r6,#0x15
    0x00000534:    e3a07016    .p..    MOV     r7,#0x16
    0x00000538:    e3a08017    ....    MOV     r8,#0x17
    0x0000053c:    e3a09018    ....    MOV     r9,#0x18
    0x00000540:    e3a0a019    ....    MOV     r10,#0x19
    0x00000544:    e59d000c    ....    LDR     r0,[r13,#0xc]
    0x00000548:    e59db008    ....    LDR     r11,[r13,#8]
    0x0000054c:    e1600b80    ..`.    SMULBB  r0,r0,r11
    0x00000550:    e59db004    ....    LDR     r11,[r13,#4]
    0x00000554:    e1600b80    ..`.    SMULBB  r0,r0,r11
    0x00000558:    e59db000    ....    LDR     r11,[r13,#0]
    0x0000055c:    e1600b80    ..`.    SMULBB  r0,r0,r11
    0x00000560:    e0000091    ....    MUL     r0,r1,r0
    0x00000564:    e0000092    ....    MUL     r0,r2,r0
    0x00000568:    e0000093    ....    MUL     r0,r3,r0
    0x0000056c:    e000009c    ....    MUL     r0,r12,r0
    0x00000570:    e000009e    ....    MUL     r0,r14,r0
    0x00000574:    e0000094    ....    MUL     r0,r4,r0
    0x00000578:    e0000095    ....    MUL     r0,r5,r0
    0x0000057c:    e0000096    ....    MUL     r0,r6,r0
    0x00000580:    e0000097    ....    MUL     r0,r7,r0
    0x00000584:    e0000098    ....    MUL     r0,r8,r0
    0x00000588:    e0000099    ....    MUL     r0,r9,r0
    0x0000058c:    e000009a    ....    MUL     r0,r10,r0
    0x00000590:    e28dd010    ....    ADD     r13,r13,#0x10
    0x00000594:    e8bd8ff0    ....    POP     {r4-r11,pc}
```

Example 34 shows a function that declares 12 local variables.

Example 34. Register Allocation with 12 Local Variables

```
int RegisterAllocationV2(void)
{
    int a=10,b=11,c=12,d=13,e=14,f=15,g=16,h=17,i=18,j=19,k=20,l=21;

    return a*b*c*d*e*f*g*h*i*j*k*l;
}
```

There are no **str** or **ldr** instructions in the assembly code shown in Example 35. The compiler allocates the 12 local variables into registers and the stack is not used. The result of the multiplications is also returned through the r0 register.

Example 35. Register Allocation with 12 Local Variables Compiler Output

```
RegisterAllocationV2
    0x00000598:    e92d47f0    .G-.    PUSH    {r4-r10,r14}
    0x0000059c:    e3a0100a    ....    MOV     r1,#0xa
    0x000005a0:    e3a0200b    ....    MOV     r2,#0xb
    0x000005a4:    e3a0300c    .0..    MOV     r3,#0xc
    0x000005a8:    e3a0c00d    ....    MOV     r12,#0xd
    0x000005ac:    e3a0e00e    ....    MOV     r14,#0xe
    0x000005b0:    e3a0400f    .@..    MOV     r4,#0xf
    0x000005b4:    e3a05010    .P..    MOV     r5,#0x10
    0x000005b8:    e3a06011    .`..    MOV     r6,#0x11
    0x000005bc:    e3a07012    .p..    MOV     r7,#0x12
    0x000005c0:    e3a08013    ....    MOV     r8,#0x13
    0x000005c4:    e3a09014    ....    MOV     r9,#0x14
    0x000005c8:    e3a0a015    ....    MOV     r10,#0x15
    0x000005cc:    e1600281    ..`.    SMULBB  r0,r1,r2
    0x000005d0:    e1600380    ..`.    SMULBB  r0,r0,r3
    0x000005d4:    e1600c80    ..`.    SMULBB  r0,r0,r12
    0x000005d8:    e000009e    ....    MUL     r0,r14,r0
    0x000005dc:    e0000094    ....    MUL     r0,r4,r0
    0x000005e0:    e0000095    ....    MUL     r0,r5,r0
    0x000005e4:    e0000096    ....    MUL     r0,r6,r0
    0x000005e8:    e0000097    ....    MUL     r0,r7,r0
    0x000005ec:    e0000098    ....    MUL     r0,r8,r0
    0x000005f0:    e0000099    ....    MUL     r0,r9,r0
    0x000005f4:    e000009a    ....    MUL     r0,r10,r0
    0x000005f8:    e8bd87f0    ....    POP     {r4-r10,pc}
```

6 Function Calls

The ARM-Thumb Procedure Call Standard defines how to pass function arguments and return values in ARM registers. Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments. For functions following the four-register rule, the compiler passes all the arguments in registers. For functions with more than four arguments, the caller and callee must access the stack for the extra arguments. Example 36 illustrates how six int arguments are allocated by the compiler.

Example 36. Function Call with Six Arguments

```
int ArgumentsAllocationV1(int a,int b,int c,int d,int e,int f)
{
    return a+b+c+d+e+f;
}
```

The generated output in Example 37 shows that the first two parameters are passed through the stack using two **ldr** instructions. The remaining arguments are passed in the r0, r1, r2 and r3 registers.

Example 37. Function Call with Six Arguments Compiler Output

```
ArgumentsAllocationV1
    0x00000194:    e92d4010    .@-.    PUSH    {r4,r14}
    0x00000198:    e1a0c000    ....    MOV     r12,r0
    0x0000019c:    e59d400c    .@..    LDR     r4,[r13,#0xc]
    0x000001a0:    e59de008    ....    LDR     r14,[r13,#8]
    0x000001a4:    e08c0001    ....    ADD     r0,r12,r1
    0x000001a8:    e0800002    ....    ADD     r0,r0,r2
    0x000001ac:    e0800003    ....    ADD     r0,r0,r3
    0x000001b0:    e080000e    ....    ADD     r0,r0,r14
    0x000001b4:    e0800004    ....    ADD     r0,r0,r4
    0x000001b8:    e8bd8010    ....    POP     {r4,pc}
```

Example 38 illustrates the benefits of using no more than four arguments.

Example 38. Function Call with Four Arguments

```
int ArgumentsAllocationV2(int a,int b,int c, int d)
{
    return a+b+c+d;
}
```

Example 39 shows that the a, b, c and d arguments are passed through the r0, r1, r2 and r3 registers and the stack is not used. The result of the sum is returned in the r0 register to the caller.

Example 39. Function Call with Four Arguments Compiler Output

```
ArgumentsAllocationV2
    0x000001bc:    e1a0c000    ....    MOV     r12,r0
    0x000001c0:    e08c0001    ....    ADD     r0,r12,r1
    0x000001c4:    e0800002    ....    ADD     r0,r0,r2
    0x000001c8:    e0800003    ....    ADD     r0,r0,r3
    0x000001cc:    e12fff1e    ../.    BX      r14
```

If a C function requires more than four arguments, it is almost always more efficient to use structures. Group related arguments into structures and pass a structure pointer rather than multiple arguments.

7 Pointer Aliasing

If two or more pointers point to the same address, then they are said to be aliased. Most of the time the compiler does not know which pointers are aliases and are not. Example 40 shows a function that increments the red, green and blue components of the same pixel by the value of offset1. The compiler must load from ColorCorrection –> offset1 three times.

Example 40. Pointer Aliasing

```
void PointerAliasingV1(Pixel *PixelValue,Coefficients *ColorCorrection)
{

    PixelValue->Red+= ColorCorrection->offset1;
    PixelValue->Green+= ColorCorrection->offset1;
    PixelValue->Blue+= ColorCorrection->offset1;
}
```

Usually the compiler optimizes the code to evaluate ColorCorrection –> offset1 once and the value is reused for the subsequent occurrences. However, in this case the compiler cannot be sure that the write to PixelValue does not affect the read from ColorCorrection. Therefore, the compiler can not use any optimizations for this case.

The assembly code in Example 41 shows two memory accesses through **ldr** instructions for the PixelValue and ColorCorrection pointers, one **add** instruction and one **str** instruction for storing the result into memory. The same assembly code pattern is repeated for the two subsequent C lines.

Example 41. Pointer Aliasing Compiler Output

```
PointerAliasingV1
    0x00000200:    e5902000    ...    LDR      r2,[r0,#0]
    0x00000204:    e5913000    .0..   LDR      r3,[r1,#0]
    0x00000208:    e0822003    ...    ADD      r2,r2,r3
    0x0000020c:    e5802000    ...    STR      r2,[r0,#0]
    0x00000210:    e5902004    ...    LDR      r2,[r0,#4]
    0x00000214:    e5913000    .0..   LDR      r3,[r1,#0]
    0x00000218:    e0822003    ...    ADD      r2,r2,r3
    0x0000021c:    e5802004    ...    STR      r2,[r0,#4]
    0x00000220:    e5902008    ...    LDR      r2,[r0,#8]
    0x00000224:    e5913000    .0..   LDR      r3,[r1,#0]
    0x00000228:    e0822003    ...    ADD      r2,r2,r3
    0x0000022c:    e5802008    ...    STR      r2,[r0,#8]
    0x00000230:    e12fff1e    ../.   BX       r14
```

Example 42 shows an example where a new local variable, localoffset, is created to hold the value of ColorCorrection –> offset1 so that the compiler performs only a single load.

Example 42. Improved Pointer Aliasing

```
void PointerAliasingV2(Pixel *PixelValue,Coefficients *ColorCorrection)
{
    int localoffset= ColorCorrection->offset1;

    PixelValue->Red+= localoffset;
    PixelValue->Green+= localoffset;
    PixelValue->Blue+= localoffset;
}
```

The assembly code in Example 43 shows how ColorCorrection –> offset1 is loaded from memory through a **ldr** instruction and held in the localoffset variable. For the subsequent localoffset references ColorCorrection –> offset1 is not loaded from memory since it is already held in the r2 register.

Example 43. Improved Pointer Aliasing Compiler Output

```
PointerAliasingV2
    0x00000234:    e5912000    ...    LDR      r2,[r1,#0]
    0x00000238:    e5903000    .0..   LDR      r3,[r0,#0]
    0x0000023c:    e0833002    .0..   ADD      r3,r3,r2
    0x00000240:    e5803000    .0..   STR      r3,[r0,#0]
    0x00000244:    e5903004    .0..   LDR      r3,[r0,#4]
    0x00000248:    e0833002    .0..   ADD      r3,r3,r2
    0x0000024c:    e5803004    .0..   STR      r3,[r0,#4]
    0x00000250:    e5903008    .0..   LDR      r3,[r0,#8]
    0x00000254:    e0833002    .0..   ADD      r3,r3,r2
    0x00000258:    e5803008    .0..   STR      r3,[r0,#8]
    0x0000025c:    e12fff1e    ../.   BX       r14
```

8 Structure Layout

Modern embedded C/C++ compilers give fine-grained control and a wealth of options for determining how C structures are laid out. The result is that any arbitrary layout can be obtained. To understand structure layout fully, first the concept of data bus width and natural boundaries are discussed.

ARM processors have a 32-bit data bus width, meaning that each memory cycle can access a maximum of 32 bits. Multi-byte quantities can be properly accessed at any address. However, if they are not properly aligned, performance is degraded because the hardware adds extra memory cycles.

Example 44 shows a structure layout.

Example 44. Structure Layout

```
typedef struct
{
    char x;
    int  y;
    char z;
    short w;

}StructV1;

void structV1()
{

    StructV1 example;

    example.x=0xaa;
    example.y=0xbbbbbbbb;
    example.z=0xcc;
    example.w=0xdddd;

}
```

For a little-endian memory system, the compiler adds padding between the structure objects to ensure that the next object is aligned to the size of that object. For Example 44,

StructV1 = {<3>x|yyyy|ww<1>z},

where <n> means the number of added padding bytes and | is used to separate data words.

The memory usage for StructV1 is three words or 12 bytes. The assembly code in Example 45 confirms the use of three data words for the StructV1 structure.

Example 45. Structure Layout Compiler Output

```
structV1
    0x00000384:    e92d400e    .@-.    PUSH    {r1-r3,r14}
    0x00000388:    e3a000aa    ....    MOV     r0,#0xaa
    0x0000038c:    e5cd0000    ....    STRB    r0,[r13,#0]
    0x00000390:    e59f02e4    ....    LDR     r0,[pc,#740]  ; [0x67c] = 0xbbbbbbbb
    0x00000394:    e58d0004    ....    STR     r0,[r13,#4]
    0x00000398:    e3a000cc    ....    MOV     r0,#0xcc
    0x0000039c:    e5cd0008    ....    STRB    r0,[r13,#8]
    0x000003a0:    e59f02d8    ....    LDR     r0,[pc,#728]  ; [0x680] = 0xffffdddd
    0x000003a4:    e1cd00ba    ....    STRH    r0,[r13,#0xa]
    0x000003a8:    e8bd900c    ....    POP     {r2,r3,r12,pc}
```

To improve the memory usage, the elements in the structure can be reordered such as in Example 46.

Example 46. Improved Structure Layout

```
typedef struct
{
    char x;
    char y;
    short z;
    int w;

}StructV2;

void structV2()
{

    StructV2 example2;

    example2.x=0xaa;
    example2.y=0xbb;
    example2.z=0xcccc;
    example2.w=0xdddddddd;

}
```

The armcc compiler aligns the size of the four objects into two data words:

StructV2{zzyx|wwww}

Example 47 shows the generated assembly code. The compiler stores 0xAA into the address [r13 + 0]. Then, the compiler stores 0xBB into address [r3 + 1]. Finally, the compiler stores 0xCCCC into address [r13+2] and 0xDDDDDDDD into address [r13 + 4].

Example 47. Improved Structure Layout Compiler Output

```
structV2
    0x000003ac:    e92d400c    .@-.    PUSH    {r2,r3,r14}
    0x000003b0:    e3a000aa    ....    MOV     r0,#0xaa
    0x000003b4:    e5cd0000    ....    STRB    r0,[r13,#0]
    0x000003b8:    e3a000bb    ....    MOV     r0,#0xbb
    0x000003bc:    e5cd0001    ....    STRB    r0,[r13,#1]
    0x000003c0:    e59f02bc    ....    LDR     r0,[pc,#700]  ; [0x684] = 0xffffcccc
    0x000003c4:    e1cd00b2    ....    STRH    r0,[r13,#2]
    0x000003c8:    e59f02b8    ....    LDR     r0,[pc,#696]  ; [0x688] = 0xdddddddd
    0x000003cc:    e58d0004    ....    STR     r0,[r13,#4]
    0x000003d0:    e8bd9008    ....    POP     {r3,r12,pc}
```

The memory is now completely aligned. It is more efficient to lay structures out in order of increasing element size. As a rule, start the structure with the smallest elements and finish with the largest.

9 Endianness

The ARM core can be configured to work in little-endian or big-endian modes. Little-endian mode is usually the default. The endianness of an ARM is usually set at power-up and remains fixed thereafter. The GetEndianness function can be used to find out the endianness at runtime as shown in Example 48.

Example 48. Endianness

```
int GetEndianness(void)
{
    long number=0xff;
    char endianness=*((char*)(&number));
    return endianness;
}
```

This function can be used to make code more portable and flexible. The function assigns 0xFF to a long variable. The code casts the variable to char and assigns it to a char variable. If the core is little-endian, the function returns 0xFF. If the core is big-endian, the function returns 0.

10 Bit-Fields

Bit-fields are structure elements and are usually accessed using structure pointers. Therefore, they suffer from pointer aliasing problems. Every bit-field access is actually a memory access and how bits are allocated within the bit-field container is compiler-dependent. Thus, bit-fields are frequently prone to portability issues. Example 49 illustrates this problem. The compiler is not able to optimize this code.

Example 49. Bit-Field

```
void change_state(void);

typedef struct
{
  unsigned char NEW:1;
  unsigned char RUNNABLE:1;
  unsigned char BLOCKED:1;
  unsigned char WAITING:1;

}state_machine;

void BitFieldsV1(state_machine *thread_state)
{

      if(thread_state->NEW)
      {
         change_state();
      }
      if(thread_state->RUNNABLE)
      {
        change_state();
      }
      if(thread_state->BLOCKED)
      {
        change_state();
      }
      if(thread_state->WAITING)
      {
        change_state();
      }

}
```

As shown in Example 50, the compiler accesses the memory containing the bit-field four times. Because the bit-field is stored in memory, the change_state function could change the value. The compiler uses two instructions to test the first if statement. For the remaining if statements the compiler uses three instructions to test the bit-fields.

Example 50. Bit-Field Compiler Output

```
BitFieldsV1
    0x00000410:    e92d4010    .@-.    PUSH    {r4,r14}
    0x00000414:    e1a04000    .@..    MOV     r4,r0
    0x00000418:    e5d40000    ....    LDRB    r0,[r4,#0]
    0x0000041c:    e3100001    ....    TST     r0,#1
    0x00000420:    0a000000    ....    BEQ     {pc} + 0x8   ; 0x428
    0x00000424:    ebfffffe    ....    BL      change_state  ;
    0x00000428:    e5d40000    ....    LDRB    r0,[r4,#0]
    0x0000042c:    e1a00f00    ....    LSL     r0,r0,#30
    0x00000430:    e1b00fa0    ....    LSRS    r0,r0,#31
    0x00000434:    0a000000    ....    BEQ     {pc} + 0x8   ; 0x43c
    0x00000438:    ebfffffe    ....    BL      change_state  ;
    0x0000043c:    e5d40000    ....    LDRB    r0,[r4,#0]
    0x00000440:    e1a00e80    ....    LSL     r0,r0,#29
    0x00000444:    e1b00fa0    ....    LSRS    r0,r0,#31
    0x00000448:    0a000000    ....    BEQ     {pc} + 0x8   ; 0x450
    0x0000044c:    ebfffffe    ....    BL      change_state  ;
    0x00000450:    e5d40000    ....    LDRB    r0,[r4,#0]
    0x00000454:    e1a00e00    ....    LSL     r0,r0,#28
    0x00000458:    e1b00fa0    ....    LSRS    r0,r0,#31
    0x0000045c:    0a000000    ....    BEQ     {pc} + 0x8   ; 0x464
    0x00000460:    ebfffffe    ....    BL      change_state  ;
    0x00000464:    e8bd8010    ....    POP     {r4,pc}
```

Example 51 implements the function using logical operations rather than bit-fields. All of the bit-fields are contained in an int type. For efficiency, a copy of their value is held in the local variable local_thread_state.

Example 51. Improved Bit-Field

```
#define NEW (1<<0)
#define RUNNABLE (1<<1)
#define BLOCKED (1<<2)
#define WAITING (1<<3)

void BitFieldsV2(int *thread_state)
{

      int local_thread_state= *thread_state;


      if(local_thread_state&NEW)
      {
         change_state();
      }
      if(local_thread_state&RUNNABLE)
      {
         change_state();
      }
      if(local_thread_state&BLOCKED)
      {
         change_state();
      }
      if(local_thread_state&WAITING)
      {
         change_state();
      }


}
```

Example 52 shows that **tst** and **beq** instructions are now used to test the if statements.

Example 52. Improved Bit-Field Compiler Output

```
BitFieldsV2
    0x00000468:    e92d4070    p@-.    PUSH    {r4-r6,r14}
    0x0000046c:    e1a04000    .@..    MOV     r4,r0
    0x00000470:    e5945000    .P..    LDR     r5,[r4,#0]
    0x00000474:    e3150001    ....    TST     r5,#1
    0x00000478:    0a000000    ....    BEQ     {pc} + 0x8  ; 0x480
    0x0000047c:    ebfffffe    ....    BL      change_state  ;
    0x00000480:    e3150002    ....    TST     r5,#2
    0x00000484:    0a000000    ....    BEQ     {pc} + 0x8  ; 0x48c
    0x00000488:    ebfffffe    ....    BL      change_state  ;
    0x0000048c:    e3150004    ....    TST     r5,#4
    0x00000490:    0a000000    ....    BEQ     {pc} + 0x8  ; 0x498
    0x00000494:    ebfffffe    ....    BL      change_state  ;
    0x00000498:    e3150008    ....    TST     r5,#8
    0x0000049c:    0a000000    ....    BEQ     {pc} + 0x8  ; 0x4a4
    0x000004a0:    ebfffffe    ....    BL      change_state  ;
    0x000004a4:    e8bd8070    p...    POP     {r4-r6,pc}
```

Logical **and**, **or** and **xor** operations with mask values reduce the overhead associated with bit-field structures. These operations compile efficiently for ARM architecture. Use #define or enum to define mask values.

11 Floating Point Versus Fixed Point

Most ARM processor implementations do not provide hardware floating point support. Because of this, the C compiler must provide support for floating-point in software. This means that the C compiler converts every floating point operation into a subroutine call.

Example 53 shows a function that combines two colors allowing for transparency effects in computer graphics. The value of alpha in the color code ranges from 0.0 to 1.0, where 0.0 represents a fully transparent color, and 1.0 represents a fully opaque color.

Example 53. Floating Point

```
char AlphaBlendingV1(char color_a,char color_b,float alpha)
{
    return (1-alpha)*color_a+alpha*color_b;
}
```

This function requires four floating point operations (two additions and two multiplications) which must be computed in software. The assembly output in Example 54 shows the compiler calls seven subroutines and the corresponding function calling overhead.

These functions are unsigned int to float conversion (__aeabi_uif2f), float multiplication (__aeabi_fmul), float subtraction (__aeabi_fsub), float add (__aeabi_fadd) and float to unsigned int conversion (__aeabi_f2uiz).

Example 54. Floating Point Compiler Output

```
AlphaBlendingV1
    0x000004a8:    e92d4ff8    .O-.    PUSH    {r3-r11,r14}
    0x000004ac:    e1a04000    .@..    MOV     r4,r0
    0x000004b0:    e1a05001    .P..    MOV     r5,r1
    0x000004b4:    e1a06002    .`..    MOV     r6,r2
    0x000004b8:    e1a00005    ....    MOV     r0,r5
    0x000004bc:    ebfffffe    ....    BL      __aeabi_ui2f   ;
    0x000004c0:    e1a0a000    ....    MOV     r10,r0
    0x000004c4:    e1a01006    ....    MOV     r1,r6
    0x000004c8:    ebfffffe    ....    BL      __aeabi_fmul   ;
    0x000004cc:    e1a08000    ....    MOV     r8,r0
    0x000004d0:    e1a00004    ....    MOV     r0,r4
    0x000004d4:    ebfffffe    ....    BL      __aeabi_ui2f   ;
    0x000004d8:    e1a0a000    ....    MOV     r10,r0
    0x000004dc:    e1a01006    ....    MOV     r1,r6
    0x000004e0:    e3a005fe    ....    MOV     r0,#0x3f800000
    0x000004e4:    ebfffffe    ....    BL      __aeabi_fsub   ;
    0x000004e8:    e1a0b000    ....    MOV     r11,r0
    0x000004ec:    e1a0100a    ....    MOV     r1,r10
    0x000004f0:    ebfffffe    ....    BL      __aeabi_fmul   ;
    0x000004f4:    e1a09000    ....    MOV     r9,r0
    0x000004f8:    e1a01008    ....    MOV     r1,r8
    0x000004fc:    ebfffffe    ....    BL      __aeabi_fadd   ;
    0x00000500:    e1a07000    .p..    MOV     r7,r0
    0x00000504:    ebfffffe    ....    BL      __aeabi_f2uiz  ;
    0x00000508:    e20000ff    ....    AND     r0,r0,#0xff
    0x0000050c:    e8bd8ff8    ....    POP     {r3-r11,pc}
```

Example 55 shows the AlphaBlendingV2 function with fixed point arithmetic. The value of alpha in the color code ranges from 0 to 255, where 0 represents a fully transparent color, and 255 represents a fully opaque color.

Example 55. Improved Floating Point

```
char AlphaBlendingV2(char color_a, char color_b,char alpha)
{
    return ( (255-alpha)*color_a + alpha*color_b )/255;
}
```

The return value of AlphaBlendingV2 is similar to AlphaBlendingV1 but far more efficient since the compiler is calling only one subroutine that performs integer division (__aeabi_idivmod) as shown in Example 56.

Example 56. Improved Floating Point Compiler Output

```
AlphaBlendingV2
    0x00000510:    e92d4070    p@-.    PUSH    {r4-r6,r14}
    0x00000514:    e1a05000    .P..    MOV     r5,r0
    0x00000518:    e1a06001    .`..    MOV     r6,r1
    0x0000051c:    e1a04002    .@..    MOV     r4,r2
    0x00000520:    e26410ff    ..d.    RSB     r1,r4,#0xff
    0x00000524:    e1610581    ..a.    SMULBB  r1,r1,r5
    0x00000528:    e1001684    ....    SMLABB  r0,r4,r6,r1
    0x0000052c:    e3a010ff    ....    MOV     r1,#0xff
    0x00000530:    ebfffffe    ....    BL      __aeabi_idivmod
    0x00000534:    e20000ff    ....    AND     r0,r0,#0xff
    0x00000538:    e8bd8070    p...    POP     {r4-r6,pc}
```

12 Conclusions

The C compiler can be facilitated to generate faster or smaller ARM code. Performance-critical applications often contain a few routines that dominate the performance profile. Code-tuning using the guidelines of this application note can improve the application performance particularly for real-time applications.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN3884
Rev. 0
07/2009