NEON™ Support in Compilation Tools

Development Article

ARM®

DHT 0004A (ID081609)

NEON Support in Compilation Tools
Development Article

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
10 August 2009	A	Non-Confidential	First release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
NEON Support in Compilation Tools Development Article

Chapter 1
NEON Support in Compilation Tools

This article describes different ways to generate NEON code using the GNU and ARM RealView® compilation tools. It contains the following sections:

1.1 Introduction

NEON technology provides *Single Instruction Multiple Data* (SIMD) operations in ARM processors implementing the Advanced SIMD architecture extensions. These operations can significantly accelerate repetitive operations on large data sets. This can be useful in applications such as media codecs.

A lot of the software development using this technology takes place in C or C++, to increase maintainability and portability, and to shorten the development cycle. This article describes how to make good use of available compilation tools to create optimized software.

1.1.1 Compilation tools

There are several sources of compilation tools for ARM platforms. This article describes the use of GNU and ARM RealView tools, but underlying concepts remain valid for any toolchain that supports NEON technology.

GNU tools

A GNU toolchain consists of different components bundled together from separate open source projects, including:

GCC C and C++ compilers.

binutils Assembler, librarian, linker and object inspection and manipulation tools.

libraries C and C++ libraries, for example glibc or uClibc.

Because these are developed by separate projects and released separately, the matrix of possible combinations is complex. The different configuration options available when building the components from sources adds more complexity. For this reason, many companies choose to use a complete bundled toolchain from a commercial provider. An alternative is to use a native GNU toolchain included in the target Linux distribution.

RealView tools

ARM supplies the *RealView Compilation Tools* (RVCT) as part of the RealView Development Suite. RVCT consists of a C and C++ compiler, an assembler, a linker, a librarian, and an image conversion utility. ARM bundles embedded C and C++ libraries with RVCT.

RVCT 4.0 introduced support for GCC command line translation. See *Cross-toolchain projects* on page 1-5 for practical examples of designing complete software projects that can be compiled with either GCC or RVCT.

1.2 Software considerations

Before you start programming, read the following sections to familiarize yourself with the fundamental aspects of NEON technology, development tools, and target platforms:

- *Reset behavior*
- *Linkage*
- *Floating-point precision*
- *Default behavior of tools* on page 1-4
- *Cross-toolchain projects* on page 1-5

1.2.1 Reset behavior

ARM processors boot with the NEON engine and *Floating-Point Unit* (FPU) disabled. Until you have properly configured and enabled these features, attempting to execute any NEON or VFP instructions results in an undefined instruction exception. On processors that implement the TrustZone Security Extensions, you might also have to explicitly permit FPU and NEON access when executing in Non-secure state. This is normally configured by your boot process and your operating system.

1.2.2 Linkage

If the target has hardware support for NEON technology or an FPU, the highest performance is achieved by passing NEON and FPU parameters and return values in NEON and FPU registers. This is called hardware floating-point linkage. In some situations, using the general-purpose registers for parameter passing might be preferred, to simplify software compatibility between platforms with and without hardware floating-point support. This is called software floating-point linkage.

You cannot mix objects with different floating-point linkage in a single image. Any dynamic libraries loaded while the application is running must also use the same linkage.

Any system that supports both NEON and VFP instructions uses a common register bank for these instructions, therefore configuration options that affect the floating-point calling convention also affect how NEON parameters are passed and returned.

1.2.3 Floating-point precision

Processors that implement the ARMv7-A architecture profile have two options for handling single-precision floating point, VFPv3 and NEON technology. VFPv3 supports full IEEE754 compliant single-precision and double-precision handling completely in hardware. The NEON engine operates on single-precision floating-point

numbers only, and its handling of denormalled numbers and NaNs is not IEEE754 compliant. The NEON engine processing of floating-point numbers is compliant with the standards of most modern programming languages, including C and C++.

1.2.4 Default behavior of tools

Compilation tools make certain assumptions about the types of instructions and calling conventions to use if these are not explicitly specified. Different tools have different default behaviors that can be overridden using, for example, command line parameters.

GNU

The GNU tools do not assume hardware floating-point support unless it is explicitly specified. You must specify `-mfpu=neon` on the command line to tell the compiler it can generate NEON instructions. You must also specify the floating-point calling convention to use, with the `-mfloat-abi` command line parameter. This takes one of the following argument:

soft No hardware floating-point support. All floating-point operations are implemented as calls to helper libraries. Soft linkage is used for floating-point function arguments and return values. When this mode is specified, the compiler does not generate NEON or VFP instructions.

softfp Soft linkage is used, but the compiler can generate hardware floating-point instructions supported by the specified floating-point unit.

hard Hard linkage is used and the compiler can generate hardware floating-point instructions supported by the specified floating-point unit.

For example, to build for NEON technology, using the soft-float calling convention, add `-mfpu=neon -mfloat-abi=softfp` to the compiler command line. To build for VFPv3 only, with no NEON instructions, specify `-mfpu=vfpv3`.

RVCT

The RealView tools assume hardware floating-point support for processors that can implement floating-point hardware unless explicitly stated otherwise. By default, they also assume hardware floating-point linkage.

For example, specifying `--cpu=Cortex-A9` on the command line permits the use of NEON and VFP instructions, with hardware linkage unless software linkage is explicitly specified.

To change the calling convention to soft linkage, you can specify, for example, `--fpu=SoftVFP+VFPv3`. This informs the tool that VFPv3 hardware is available but that it must use the software calling convention.

Build 591 of RVCT 4.0 introduced support for specifying floating-point linkage independently from the hardware support. The parameters `--apcs=/softfp` and `--apcs=/hardfp` specify soft and hard linkage respectively.

1.2.5 Cross-toolchain projects

RVCT 4.0 introduced GCC command line translation using the `--translate-gcc` and `--translate-g++` parameters. This makes it possible to design a software project to be compiled with either GCC or RVCT without duplicating the build system. Because of differences in syntax between the toolchains, the inline or embedded assemblers cannot be used. You must place any assembly language used in the project in standalone assembly language modules. Because the syntaxes of the standalone assemblers also differ, you must then select which assembler to use consistently throughout the project.

1.3 Intrinsics

Intrinsic functions and data types, or intrinsics in the shortened form, provide access to low-level NEON functionality from C or C++ source code. They use syntax that is similar to function calls. Software can pass NEON vectors as function arguments or return values, and declare them as normal variables.

Intrinsics provide almost as much control as writing assembly language, but leave the allocation of registers to the compiler, so that you can focus on the algorithms. Also, the compiler can optimize the intrinsics like normal C or C++ code, replacing them with more efficient sequences if possible. It can also perform instruction scheduling to remove pipeline stalls for the specified target processor. This leads to more maintainable source code than using assembly language.

Example 1-1 shows a short function that takes a four-lane vector of 32-bit unsigned integers as input parameter, and returns a vector where the values in all lanes have been doubled.

Example 1-1 Using NEON intrinsics in C code

```
#include <arm_neon.h>

uint32x4_t double_elements(uint32x4_t input)
{
    return(vaddq_u32(input, input));
}
```

Example 1-2 shows the disassembled version of the code generated from Example 1-1, compiled for hardware linkage. The `double_elements` function translates to a single NEON instruction and a return sequence.

Example 1-2 Disassembly of instructions generated by intrinsics example

```
double_elements PROC
    VADD.I32 q0,q0,q0
    BX       lr
    ENDP
```

Example 1-3 on page 1-7 shows the disassembly of the same example compiled for software linkage. In this situation, the code must copy the parameters from general-purpose registers to a NEON vector register before use. After the calculation, it must copy the return value back from NEON registers to general-purpose registers.

Example 1-3 Disassembly of instructions generated by intrinsics example

```
double_elements PROC
    VMOV     d0,r0,r1
    VMOV     d1,r2,r3
    VADD.I32 q0,q0,q0
    VMOV     r0,r1,d0
    VMOV     r2,r3,d1
    BX       lr
    ENDP
```

GCC and armcc support the same intrinsics, so code written with NEON intrinsics is completely portable between the toolchains. There are no specific command line options required for the compiler to process NEON intrinsics. You must include the `arm_neon.h` header file in any source file using intrinsics, and must specify the command line options described in *Default behavior of tools* on page 1-4.

It can be useful to have a source module optimized using intrinsics, that can also be compiled for processors that do not implement NEON technology. The macro `__ARM_NEON__` is defined by gcc when compiling for a target that implements NEON technology. RVCT 4.0 build 591 or later also define this macro. Software can use this macro to provide both optimized and plain C or C++ versions of the functions provided in the file, selected by the command line parameters you pass to the compiler.

For information about the intrinsic functions and vector data types, see the:

- RealView Compilation Tools Compiler Reference Guide, available from `http://infocenter.arm.com`
- GCC documentation, available from `http://gcc.gnu.org/onlinedocs/gcc/ARM-NEON-Intrinsics.html`

1.4 Automatic vectorization

Instead of using intrinsics to take advantage of the NEON technology in your target processor, you can use a compiler that automatically vectorizes your standard C or C++ code. This can give access to NEON performance without requiring assembler or intrinsics programming.

Both GCC and RVCT support automatic vectorization for NEON technology, but because the C and C++ standards do not cover the concurrency aspects, you might have to provide the compiler with additional information to get full benefit. The required source code modifications are part of the standard language specifications, so they do not affect code portability between different platforms and architectures.

This section describes how to enable automatic vectorization for your compiler, and small modifications to your source code that can lead to great improvements in vectorized software performance.

1.4.1 GNU tools

To enable NEON vectorization, specify `-ftree-vectorize` and `-mfpu=neon` on the compiler command line. Compiling at optimization level `-O3` implies `-ftree-vectorize` is implicit. `-ftree-vectorize` is not an ARM-specific option, it is available for many architectures that support SIMD operations.

To get more information about the vectorizations the compiler is performing, or is unable to perform because of possible dependencies, specify `-ftree-vectorizer-verbose` on the command line. This parameter takes an integer value specifying the level of detail to provide, where 1 enables additional printouts and higher values add even more information.

1.4.2 RealView tools

RVDS 4.0 Professional includes support for the vectorizing compiler. This support is enabled when you do any of the following:

- compile for maximum performance, with `-Otime`
- use aggressive optimizations, with `-O2` or `-O3`
- specify the `--vectorize` command line parameter.

Specify the armcc command line parameter `--remarks` to provide more information about the optimizations performed, or problems preventing the compiler from performing certain optimizations.

1.4.3 Optimizing for vectorization

The C and C++ languages do not provide syntax that specifies concurrent behavior, so compilers cannot safely generate parallel code. However, the developer can provide additional information to let the compiler know where it is safe to vectorize.

Unlike intrinsics, these modifications are not architecture dependant, and are likely to improve vectorization on any target platform, normally without any negative impact on performance on targets where vectorization is not possible.

The following sections describe the key rules:

- *Indicate knowledge of number of loop iterations*
- *Avoid loop-carried dependencies* on page 1-10
- *Use the restrict keyword* on page 1-10
- *Avoid conditions inside loops* on page 1-11
- *Use suitable data types* on page 1-11.

Indicate knowledge of number of loop iterations

If a loop has a fixed iteration count, or if you know that the iteration count is always an even multiple, indicating this can permit the compiler to perform optimizations that would otherwise be unsafe.

Example 1-4 shows a function accumulating `len` number of `int`-sized elements. If you know that the value passed as `len` is always a multiple of four, you can indicate this to the compiler by masking off the bottom two bits when comparing the loop counter to `len`. Because this loop now always executes a multiple of four times, the compiler can safely vectorize it.

Example 1-4 Indicating known iteration multiple

```
int accumulate(int * c, int len)
{
    int i, retval;

    for(i=0, retval = 0; i < (len & ~3) ; i++) {
        retval += c[i];
    }

    return retval;
}
```

Avoid loop-carried dependencies

If your code contains a loop where the result of one iteration is affected by the result of a previous iteration, this prevents the compiler from vectorizing it. If possible, restructure the code to remove any loop-carried dependencies.

Use the restrict keyword

C99 introduced the `restrict` keyword, that you can use to inform the compiler that the location accessed through a specific pointer is not accessed through any other pointer within the current scope.

Example 1-5 shows a situation where using `restrict` on a pointer to a location being updated makes vectorization safe when it otherwise would not be.

Example 1-5 Use of the restrict keyword

```
int accumulate2(char * c, char * d, char * restrict e, int len)
{
    int i;

    for(i=0 ; i < (len & ~3) ; i++) {
        e[i] = d[i] + c[i];
    }

    return i;
}
```

Without the keyword, the compiler must assume that `e[i]` can refer to the same location as `d[i + 1]`, meaning that the possibility of a loop-carried dependency prevents it from vectorizing this sequence. With `restrict`, the programmer informs the compiler that any location accessed through `e` is only accessed through pointer `e` in this function. This means the compiler can ignore the possibility of aliasing and vectorize the sequence.

Note

Using `restrict` does not change the function prototype in any externally-visible way. If you pass to this function values for `c` or `d` that might overlap with `e`, the vectorized code might not execute correctly.

Both GCC and RVCT support the alternative forms `__restrict__` and `__restrict` when not compiling for C99. RVCT also supports using the `restrict` keyword with C90 and C++ when `--restrict` is specified on the command line.

Avoid conditions inside loops

Normally, the compiler cannot vectorize loops containing conditional sequences. In the best case, it duplicates the loop, but in many cases it cannot vectorize it at all.

Use suitable data types

When optimizing some algorithms operating on 16-bit or 8-bit data without SIMD, sometimes you get better performance if you treat them as 32-bit variables. When producing software targeting automatic vectorization, for best performance always use the smallest data type that can hold the required values. In a given period, the NEON engine can process twice as many 8-bit values as 16-bit values.

Also, NEON technology does not support some data types, and some are only supported for certain operations. For example, it does not support double-precision floating-point, so using a double-precision `double` where a single-precision `float` is sufficient can prevent the compiler from vectorizing code. NEON technology supports 64-bit integers only for certain operations, so avoid using `long long` variables where possible.

Note

NEON technology includes a group of instructions that can perform structured load and store operations. These instructions can only be used for vectorized access to data structures where all members are of the same size.

GCC 4.4 does not support vectorization with varying vector sizes. By default, it vectorizes for doubleword registers only. You can instruct `gcc` to vectorize for quadword registers instead by specifying `-mvectorize-with-neon-quad` on the command line.

1.4.4 Floating-point vectorization

Some floating-point operations are not vectorized by default, because this can result in a loss of precision if the programmer has sorted the input values to achieve maximum precision. If the algorithm does not require this level of precision, you can specify `--fpmode=fast`, for `armcc`, or `-ffast-math`, for `gcc`, on the command line to enable these optimizations.

Example 1-6 on page 1-12 shows a sequence that can only be vectorized with one of these parameters specified. In this case, it performs parallel accumulation, potentially reducing the precision of the result.

Example 1-6 Floating-point loop requiring additional parameters to vectorize

```
float g(float const *a)
{
    float r = 0;
    int i;

    for (i = 0 ; i < 32 ; ++i)
        r += a[i];

    return r;
}
```

NEON technology always operates in Flush-to-Zero mode, making it non-compliant with IEEE754. By default, armcc uses `--fpmode=std`, permitting the deviations from the standard. However, if the command line parameters specify a mode option requiring IEEE754 compliance, for example `--fpmode=ieee_full`, most floating-point operations cannot be vectorized.

Appendix A
Implementing a NEON function using vectorization, intrinsics, or assembly language.

This appendix shows the `omxSP_DotProd_S16` function, defined in the OpenMAX Development Layer specification, implemented in three different ways, all using the NEON technology in an ARMv7-A processor.

Note

OpenMAX is a royalty-free cross platform API standard created and distributed by the Khronos Group.

A.1 Different implementations of the omxSP_DotProd_S16 function

The omxSP_DotProd_S16 function calculates the dot product of two supplied arrays of signed 16-bit integers. The arrays must be 8-byte aligned in memory. It returns the result as a signed 32-bit integer. Internal accumulators must be at least 32-bits in size.

A.1.1 Vectorized C code

Example A-1 shows how you can implement the omxSP_DotProd_S16 function in C code. Because this function does not modify any locations accessed through pointers, you do not have to use the restrict keyword to permit vectorization. Both GCC and RVCT can vectorize this code.

Example A-1 Plain C implementation

```
#include "omxtypes.h"
#include "armOMX.h"
#include "omxSP.h"

OMX_S32 omxSP_DotProd_S16(const OMX_S16 *pSrc1,
                          const OMX_S16 *pSrc2,
                          OMX_INT len)
{
    OMX_S32 retval = 0;

    while(len != 0) {
        /* 32-bit accumulation, so use 32-bit temporaries */
        OMX_S32 Var1, Var2;

        len--;
        Var1 = pSrc1[len];
        Var2 = pSrc2[len];

        retval += Var1 * Var2;
    }

    return retval;
}
```

A.1.2 C with intrinsics

Example A-2 shows how you can implement the algorithm in Example A-3 on page A-5 using NEON intrinsics. You can use either GCC or RVCT to compile the example, but the resulting assembler output might differ, because the compilers apply different optimizations.

Example A-2 Intrinsics implementation

```
#include <arm_neon.h>

#include "omxtypes.h"
#include "armOMX.h"
#include "omxSP.h"

static const OMX_S16 MaskTable16[] =
{
    0xFFFF, 0xFFFF, 0xFFFF, 0x0000, 0x0000, 0x0000
};

OMX_S32 omxSP_DotProd_S16(const OMX_S16 *pSrc1,
                          const OMX_S16 *pSrc2,
                          OMX_INT len)
{
    int32x4_t qAccumulator;
    OMX_S32 retval = 0;

    /* Initialize all lanes in accumulation vector to zero */
    qAccumulator = vdupq_n_s32(0);

    /* Process elements, 4 in parallel */
    while(len >= 4) {
        int16x4_t dVect1, dVect2;

         /* Load data from source buffers into calculation vectors */
        dVect1 = vld1_s16(pSrc1);
        dVect2 = vld1_s16(pSrc2);

        /* Multiply 16-bit lanes in dVect1 with corresponding lanes in dVect2
         * Add resulting products to 32-bit lanes in accumulator vector */
        qAccumulator = vmlal_s16(qAccumulator, dVect1, dVect2);

        /* Update pointers and size counter */
        pSrc1 += 4;
        pSrc2 += 4;
        len -= 4;
    }
```

```
    /* Handle any remaining (1-3) elements */
    if(len != 0) {
        int16x4_t dVect1, dVect2, dMask;

        /* Load vectors - this is safe even though we might be accessing beyond
         * buffers described by pSrc1/pSrc2 because these are guaranteed to be
         * 8-byte aligned and we are loading 8-byte values */
        dVect1 = vld1_s16(pSrc1);
        dVect2 = vld1_s16(pSrc2);

        /* Mask off superfluous elements in dVect1 (subsequent multiplication
         * eliminates them in dVect2) */
        dMask = vld1_s16(&MaskTable16[3 - len]);
        dVect1 = vand_s16(dVect1, dMask);

        /* Accumulate remaining values */
        qAccumulator = vmlal_s16(qAccumulator, dVect1, dVect2);
    }

    /* Accumulate the lanes in the accumulation vector qAccumulator into an
     * OMX_S32 return value */
    {
        int32x2_t dAccL, dAccH;

        /* Split 128-bit qAccumulator into 64-bit dAccL and dAccH for
         * accumulation */
        dAccL = vget_low_s32(qAccumulator);
        dAccH = vget_high_s32(qAccumulator);

        /* Accumulate 2 lanes in dAccL and dAccH into 2 lanes in dAccL */
        dAccL = vadd_s32(dAccL, dAccH);
        /* Accumulate 2 lanes in dAccL into first (and second) lane of dAccL */
        dAccL = vpadd_s32(dAccL, dAccL);

        /* Add accumulated value to retval */
        retval = vget_lane_s32(dAccL, 0);
    }

    return retval;
}
```

A.1.3 Assembly language implementation

Example A-3 on page A-5 shows the implementation of `omxSP_DotProd_S16`. This example is included for reference. Because this article is about the use of compilers, it does not explain the assembly language code. The example accesses an external lane mask table that is not shown.

Example A-3 Assembly implementation

```
    IMPORT armCOMM_qMaskTable16
    EXPORT omxSP_DotProd_S16
omxSP_DotProd_S16 PROC
    VMOV.I16  d0,#0
    VMOV.I32  q1,#0
    LDR       r3,=armCOMM_qMaskTable16
dotProdVectorLoop
    VMLAL.S16 q1,d0,d1
    VLD1.16   {d0},[r0]!
    SUBS      r2,r2,#4
    VLD1.16   {d1},[r1]!
    BGE       dotProdVectorLoop
    ADDS      r2,r2,#4
    ADDGE     r3,r3,r2,LSL #4
    VLD1.16   {d4},[r3]!
    VAND      d0,d0,d4
    VMLAL.S16 q1,d0,d1
    VADD.I32  d2,d2,d3
    VPADD.I32 d2,d2,d2
    VMOV.32   r0,d2[0]
    BX        lr
    ENDP
```

Appendix B
Revisions

This appendix describes the technical changes between released issues of this book.

Table B-1 Issue A

Change	Location	Affects
First release	-	-