User's Guide

Application Software Optimization on the C29 CPU

ABSTRACT

This user's guide describes specific techniques to optimize application performance with the C29 CPU.

Table of Contents

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

This guide describes specific techniques to optimize application performance with the C29 CPU. The advocated methods span compiler settings, memory configuration, code construction and configuration, and finally application level optimization.

2 Performance Optimization

2.1 Compiler Settings

This section discusses key compiler settings that affect performance.

2.1.1 Enabling Debug and Source Inter-Listing

During initial development, it is recommended to use the -g compiler option to generate debug information.

Note

In some cases, loops (for example, FIR) can be less performant when debug information is generated (the -g option). This is expected to be addressed in a future release of the compiler.

Then, with the following command, the output executable can generate a disassembly file with inter-listed source code. For more information, see Development Flow Differences.

```
c29objdump --disassemble -S <>.out > <>.cdis
```

2.1.2 Optimization Control

-O3 optimization is recommended for speed, specifically for software pipe-lining of loops. For debug purposes, optimization can be selectively switched off for select functions using the 'optnone' attribute.

```
__attribute__((optnone))
void foo()
{
   ..
}
```

2.1.3 Floating-Point Math

-ffast-math is a compiler option that is recommended for floating-point computations. This option lets the compiler make aggressive assumptions about floating-point math, resulting in limited loss in accuracy. Details on controlling floating-point behavior can be found in the Clang Compiler User's manual and in the C29 Clang Compiler Tools User's Guide. Details on -ffast-math can also be found in the Clang Compiler User's manual and in the C29 Clang Compiler Tools User's Guide.

With -ffast-math, the compiler replaces calls to many standard RTS functions with the corresponding TMU instruction. The TMU is built into the C29 CPU.

Note

C29 Compiler 1.0.0.LTS does not yet implement the above optimizations to RTS libraries, except for sinf() and cosf(). The next release 2.0.0.STS is expected to include these optimizations.

Single-precision floating-point division using the C '/' operator is implemented using PREDIVF, SUBC4F (7 times), and POSTDIVF instructions. Double-precision floating-point division using the C '/' operator is implemented using PREDIVF, SUBC3F (19 times), and POSTDIVF instructions. With the -ffast-math compiler option, single-precision floating-point division is implemented using the DIVF instruction (which estimates the reciprocal of the denominator and multiplies with the numerator).

2.1.4 Fixed-Point Division

In signed or unsigned 32-bit or 64-bit integer division, the C '/' operator is implemented by the compiler using the necessary instructions. Three types of division are supported - traditional, Euclidean, and Modulo. Traditional division is natively supported by the C standard and compiler, where the remainder has the sign of the numerator. In modulo (or floored) division, the remainder has the sign of the denominator. Euclidean division is the preferred choice for control operations, where the quotient is linear about 0, and the remainder is always positive.

Note

To implement Euclidean or Modulo division, intrinsics need to be used. For more information, see here.

2.1.5 Single vs Double Precision Floating-Point

If FPU64 is available (CPU3 on F29H85x), double-precision floating point operations can be efficiently performed. To enable use of the FPU64, use the compiler option:

```
-mfpu=f64
```

On the C29, EABI is the only supported executable format. COFF is not supported. With EABI, the double type is 64-bits. User code that contains literal constants (1.54) without a trailing 'f' (1.54f) is interpreted as double precision per the C standard. This leads to implicit conversion of other associated variables to double precision, which negatively impacts performance when FPU64 is not available (CPU3 on F29H85x).

Using the following compiler option generates a warning when the above occurs:

```
-Wdouble-promotion
```

2.1.6 Link-Time Optimization (LTO)

The compiler tools, starting with version 2.0.0.STS, are expected to enable inter-module optimization over a whole-program at link-time. This feature is commonly referred to as *Link-Time Optimization* or *LTO.* This can result in significant performance benefits. More information on LTO can be found here.

Note

In order for libraries included in an application to participate in LTO, the libraries must each be built with the -flto compiler option. This sets the libraries up to support LTO during linking.

2.2 Memory Settings

This section discusses key memory settings that affect performance.

2.2.1 Executing Code From RAM

To understand which RAMs are 0 wait-state access for Program code for C29 CPUs, see the *Memory Subsystem (MEMSS)* chapter of the *F29H85x and F29P58x Real-Time Microcontrollers Technical Reference Manual*. As an example, CPU1 and CPU2 have 0-WS access for program code on LPAx RAM. CPU1 and CPU3 have 0-WS access for program code on CPAx RAM.

Functions that need to execute from RAM can be placed in a RAM section, and the linker command file can be used to control the copy of this function to RAM at boot time. More information can be found here.

```
Source file:
__attribute__((section("ramfunc"), noinline)) void foo() {.. }
```

```
Linker command file:
ramfunc : load=FLASH, run=RAM, table(BINIT)
```

2.2.2 Executing Code From Flash

To understand the number of required Flash wait states depending on the CPU clock frequency, see the *Flash Parameters* section of the *F29H85x and F29P58x Real-Time Microcontrollers Data Sheet*. Also, make sure Pre-fetch, Pre-read and caches are enabled. Flash_initModule() can be used to perform these operations:

```
voidFlash_initModule(uint16_twaitstates)
{
   ..
   // Set waitstates according to frequency
   Flash_setWaitstates(waitstates);
   ..
   // Enable data cache, code cache, prefetch, and data preread to    improve performance of code//
executed from flash.
   Flash_configFRI(FLASH_FRI1, FLASH_DATAPREREAD_ENABLE | FLASH_CODECACHE_ENABLE |
FLASH_DATACACHE_ENABLE | FLASH_PREFETCH_ENABLE);
   ..
}
```

2.2.3 Data Placement

To understand which RAMs are 0 wait-state access for Program data for C29 CPUs, see the *Memory Subsystem (MEMSS)* chapter of the *F29H85x and F29P58x Real-Time Microcontrollers Technical Reference Manual*. As an example, CPU1 and CPU2 have 0-WS access for program data on LDAx RAM. CPU1 and CPU3 have 0-WS access for program data on CDAx RAM.

Parallel accesses to RAM can lead to arbitration and results in stalls when they occur to the same RAM block (LDAx, CDAx - each "x" corresponds to a different RAM block). The compiler attempts to perform parallel loads whenever feasible:

```
LD.32     M2,*(ADDR2)(A7++)
||LD.32     M3,*(ADDR2)(A4+A0<< 2)
```

To avoid stalls, make sure the accesses occur to different blocks. For example, with an FIR filter, parallel loads of filter coefficients and history buffer values can occur. Place each in its own RAM block.

2.3 Code Construction and Configuration

This section discusses code construction and configuration that affect performance.

2.3.1 Inlining

Inlining can lead to performance benefits by eliminating the overhead of function calls and returns on very small functions, allowing the compiler to perform optimizations in the context of the code surrounding the function calls. It can also be beneficial on large functions that are called only a few times.

To enable inlining, the compiler needs the optimization level to -O1 or above (at -O0, attributes can force inlining), and needs to be able to see the definition of functions at compile time. Thus, calls to functions defined in the same source file can be inlined, as well as functions defined with "static" in header files, where the header files are included in the source file.

Note

Compiler updates enabling Link-time Optimization (LTO) enable it to inline functions that are defined in source files different from the source files where they are called from.

2.3.2 Intrinsics

The C29 has instructions that support the efficient implementation of many standard RTS functions. As previously mentioned, other than sinf() and cosf(), this mapping is not yet implemented in the compiler, and is expected to be supported in the 2.0.0.STS release of the compiler when the right compiler options are used.

The compiler also supports builtins, or intrinsics, that correspond to these instructions. These intrinsics can be used today, and the F29-SDK provides examples in examples/rtlibs/fastmath/tmu that illustrate the use of these intrinsics. The examples supported are asinf(), acosf(), atan2f(), ceilf(), cosf(), divf(), expf(), floorf(), fmodf(), roundf(), sinf(), and truncf().

- Additionally, the compiler is expected to support sqrtf() and 1/sqrtf() implementation with the ISQRTF instruction. The corresponding intrinsic is shown in the code block below.

```
float __builtin_c29_i32_isqrtf32_m(float x);
```

- The intrinsic for IEXP2F is shown in the code block below.

```
float __builtin_c29_i32_iexp2f32_m(float f0);
```

Note

Due to accuracy limitations with IEXP2F when the base or exponent are large, the compiler does not support expf(), exp2f(), 1/expf(), or 1/exp2f() implementation through IEXP2F. However, the F29x-SDK contains an example for this in examples/rtlibs/fastmath/tmu/ccs/expf_example, that demonstrates the use of the intrinsic over a range of inputs. The instruction (and intrinsic) are still accurate and useful over a range of base/exponent values.

- The intrinsic for LOG2F is shown in the code block below. It can be used to implement logf(), log2f(), and also powf().

```
float __builtin_c29_i32_log2f32_m(float f0);
```

- atanf() can be implemented with PUATANF, using the intrinsic float __builtin_c29_i32_puatanf32_m(float f0)

```
Example:
// x is per-unit in [-1,1]
// y is per-unit in [-0.125, 0.125] i.e. [-pi/4, pi/4] radians
y = __builtin_c29_i32_puatanf32_m(x);
```

- atan2f() can be implemented with with PUATANF and QUADF, using the intrinsics float __builtin_c29_i32_puatanf32_m(float f0) and float __builtin_c29_quadf32(unsigned int * tdm_w_uip0, float * rw_fp1, float * rw_fp2)

```
Example:
test_output =puatan2f32(y_input,x_input);

static inline float32_t puatan2f32(float32_t y, float32_t x)
{
   uint32_t flags;
   return __builtin_c29_quadf32(&flags, &y, &x) +      __builtin_c29_i32_puatanf32_m(y / x);
}
```

2.3.3 Volatile Variables

The 'volatile' keyword on a variable indicates to the compiler that it might be modified by something external to the obvious flow of the program such as an ISR. This makes sure the compiler preserves the number of reads and writes to the global variable exactly as written in C/C++ code, without eliminating redundant reads or writes or re-ordering accesses. The volatile keyword must be used when accessing memory locations that represent memory mapped peripherals.

Note

The volatile keyword is recommended on variables only when absolutely needed, such as variables updated inside ISRs, and memory mapped peripherals. When using volatile data types, performance can be improved by using local variables for intermediate computation instead of directly referencing the volatile data structure.

2.3.4 Function Arguments

When pointers are passed as function arguments, using the "restrict" keyword on the pointer can result in performance improvements. By applying restrict to the type declaration of a pointer **p**, the programmer provides the following to the compiler:

Within the scope of the declaration of **p** , only **p** or expressions based on **p** are used to access the object pointed to by **p**.

The compiler can take advantage of this to generate more efficient code.

```
Example:
void matrix_vector_product(float32_t *restrict A, float32_t *restrict b, int nr, int nc, float32_t
*restrict c)
{
   int i, j;
   float32_t s;
   for(i = nr -1; i >=0; i--)
   {
     s =c[i];
     for(j = nc -1; j >=0; j--)
     {
        s = s +A[j*nr+i]*b[j];
     }
     c[i] = s;
    }
}
```

Note

When passing a structure as a function argument, passing structure pointers instead of structure members results in improved performance.

2.3.5 Enabling 64-Bit Data Accesses

Data buses on the F29x are 64-bit wide, allowing for 64-bit data reads and writes. However, most user code is limited to 32-bit data, and therefore accesses are limited to 32-bits. In some cases, especially array accesses, significant performance benefits can be achieved by re-writing the code to perform 64-bit accesses instead of 32-bit accesses. For example, the first code block below represents a simple memory buffer read operation, through 32-bit accesses.

```
uint32_t mem_read_16k_cn(uint32_t *src)
{
    uint32_t i =0;
    uint32_t x =0;
    for (i=0;i<LEN_16K;i++)
    {
        x +=*src++;
    }
    return x;
}
```

The next code block represents the identical operation, implemented using 64-bit accesses, which is twice as efficient.

```
uint32_t mem_read_16k_opt(uint32_t *src)
{
    uint32_t i =0;
    uint64_t *s2 = (uint64_t*) src;
    uint32_t x =0;
    uint32_t x2 =0;
    for (i=0;i<LEN_16K>>1;i++)
    {
        uint64_t temp =*s2++;
        x += (temp>>32);
        x2 += (temp&0xFFFFFFFF);
    }
    return x+x2;
}
```

2.3.6 Auto Code-Generation Tools

Performance optimization with auto code-generation tools, Mathworks Embedded Coder, for example, is a key focus area. Details can be found here.

Depending on the configuration settings, generated code can have double-precision floating point operations and these can lead to significant deterioration of performance when running on floating-point hardware that supports only single-precision floating point operations. It is advised that users look for the following in the generated code:

- Any unexpected double-precision floating point constants in C code. These can be floating point numbers without a trailing 'f'. Sometimes these have specific names, such as "DBL_EPSILON".
- Ocurrences of double-precision operations in compiler generated assembly. These might have specific names, such as "CALLD @__extendsfdf2" (double-precision multiply), or "CALLD @__muldf3" (double-precision multiply).

2.3.7 Accurately Profiling Code

The user can use different profiling techniques to benchmark their application code. With optimization enabled, the compiler can re-order operations and perform inlining, and so forth. This can sometimes make it difficult to understand exactly what was profiled, and whether what was profiled was indeed what the user wanted to profile.

"__builtin_instrumentation_label()" is a helpful label to use before and after the code that needs to be profiled. It allows for accurate profiling with optimization enabled, since the compiler does not perform re-ordering of instructions across this label. For example:

```
// Example 1
__builtin_instrumentation_label("profiling_start");
function1();
__builtin_instrumentation_label("profiling_stop");

// Example 2
__builtin_instrumentation_label("profiling_start");
// code being profiled
..
..
__builtin_instrumentation_label("profiling_stop");
```

2.4 Application Code Optimization

This section discusses application code and its configuration that affects application performance.

2.4.1 Optimized SDK Libraries

Use optimized libraries and source provided in F29x SDKs. These contain optimal implementations of many standard control, DSP, and math operations. Some of these (FFT, FIR) are written in assembly.

Many RTS library functions are cycle intensive because they cover all corner-case scenarios. When certain assumptions are made (for example, no NaN or infinite values are operands or results of floating-point operations), these functions can be replaced with simpler and more optimized functions that leverage specific C29 instructions. For example- asinf(), acosf(), atan2f(), ceilf(), cosf(), divf(), expf(), floorf(), fmodf(), roundf(), sinf(), truncf(). Examples of these implementations are provided in F29x SDKs, and are enabled with the -ffast-math compiler option.

Note

C29 Compiler 1.0.0.LTS does not yet implement the above optimizations to RTS libraries, except for sinf() and cosf(). The next release is expected to include these optimizations.

Automotive applications using AUTOSAR leverage math libraries generated by code generation tools, containing floating-point and fixed-point libraries, with functions for fixed-point to floating-point conversion and vice versa. C29 instructions can be leveraged to do these in an efficient manner.

Note

Optimized functions for AUTOSAR specific auto-generated math libraries are planned.

2.4.2 Optimizing Code-Size With Libraries

Applications can include pre-compiled libraries, but can not use all the functions present in those libraries. To make sure the linker excludes unused functions in libraries and application code, make sure both are built with the below compiler option:

```
-ffunction-sections
```

Each function is then placed in a specific section, like .text.<function_name>, otherwise each function is placed in .text.

Note

Using __attribute__((section)) puts code into the corresponding section. -ffunction-sections does not affect objects with the section attribute. In such cases, if the user groups multiple functions into the same section, even if one of those functions is used, all the functions mapped to that section are linked into the final executable.

2.4.3 C29 Special Instructions

C29 supports a number of instructions that find use in optimizing specific types of functions. Key examples are listed below:

- SVGEN - space vector generation can be optimized with the QUADF instruction, leveraged using the intrinsic 'float __builtin_c29_quadf32(unsigned int * tdm_w_uip0, float * rw_fp1, float * rw_fp2)'.

Note

Optimized implementations of the SVGEN (including those that can be applied to a 3-level inverter) are planned in the F29x Motor Control SDK.

- CRC - Cyclic Redundancy Check implementations can be optimized with the CRC instruction. Examples are provided in the F29x SDK. An intrinsic is also available 'unsigned int __builtin_c29_i32_crc(unsigned int ui0, unsigned int ui1, unsigned int ui2, unsigned int ui3)'.
- Limiting (Saturation) operations - can be optimized using the MINMAXF instruction.
  - The compiler generates the MINMAXF instruction and an excellent implementation if the C code is written using the ternary operator, if min and max are constants, and the code is compiled with -O3 and -ffast-math options. This implementation is the fastest as it involves 2 MV instructions in parallel, followed by the MINMAXF instruction. If min and max are constants, but only -O3 is used, MINMAXF is not generated, instead CMPF and SELECT instruction pairs are generated. If min and max are not constant, with -O3 and -ffast-math options, the compiler generates LDs to read from memory, CMPF, SELECT, and MAXF.

```
float saturation(float in)
{
    float out;
    out = (in > max)? max:((in < min)? min:in);
    return out;
}
```

– With if.else conditionals, the behavior is exactly the same as above.

```
float saturation(float in)
{
   float out;
   if(in > max) {
      out = max;
   } else if(in < min)
   {
      out = min;
   } else {
      out = in;
   }
   return out;
}
```

– However, the compiler does not generate the MINMAXF instruction, and a sub-optimal implementation if the C code is written with if..else conditionals and an empty (or absent) else { } statement.

```
float saturation(float in)
{
   float out = in;
   if(in > max) {
      out = max;
   } else if(in < min)
   {
      out = min;
   } else {
   }
   return out;
}
```

- Deadzone operations - in this case, the structure of the code does matter. The most efficient code is generated when min and max are constants, with the -O3 option, and with the C code written using the ternary operator as shown in the code block below. With if..else based C code, optimal code is not generated.

```
float deadzone(float in)
{
   float out;
   out = (in>1.0f)?(in-1.0f):((in>-1.0f)?0.0f:(in+1.0f));
   return out;
}
```

2.4.4 C29 Parallelism

- The C29 compiler can leverage the parallelism of the C29 architecture, executing multiple instructions in parallel especially in cases where independent operations occur sequentially. For example, the code block below demonstrates two identical PID operations that occur sequentially. If DCL_runPID is declared as a static function in a header file, the compiler can perform inlining and then perform the two PID operations in parallel.

```
float run_dualPID(DCL_PID *restrict p1, DCL_PID *restrict p2,float32_t rk1, float32_t yk1,
float32_t lk1,float32_t rk2, float32_t yk2, float32_t lk2)
{
   float x = DCL_runPID_C3(p1, rk1, yk1, lk1);
   float y = DCL_runPID_C3(p2, rk2, yk2, lk2);
   return x+y;
}
```

- Binary LUT search - binary look-up table searches are common in motor control applications, and can be optimized by changing the conditional loop to a fixed iteration loop.

Note

Optimized implementation of the Binary LUT search is expected to be released in F29-SDK 1.01.00

2.4.5 32-Bit Variables and Writes Preferred

The ECC bits cover 32-bit data, so for write sizes less then 32-bits to RAM , the memory wrapper performs a Read-Modify-Write operation to patch in the new value and re-calculate the ECC for the whole 32-bit word. This leads to stalls when multiple writes of less than 32-bits occur. This is true for most CPUs, including ARM CPUs.

```
Example: 5 writes take 13 cycles
ST.16 *(ADDR1)(A4+#0x1a),#0x1
ST.16 *(ADDR1)(A4+#0x14),#0x303
ST.8 *(ADDR1)(A4+#0x1e),#0x0
ST.8 *(ADDR1)(A4+#0x16),#0x4
ST.16 *(ADDR1)(A4+#0x1c),#0x0
```

Note

Application code must minimize writes of less than 32-bits, and in general use 32-bit variables where possible.

Using 32-bit variables also sometimes avoids the compiler adding extra instructions to sign extend 16-bit values. The below example shows an additional instruction the compiler uses to sign-extend a 16-bit value to a 32-bit value.

```
Example:
int16_t mashup_16(int16_t in_a, int16_t in_b)
{
int16_t tmp1, tmp2, tmp3, tmp4;
   tmp1 = in_a + in_b;
   tmp2 = in_a - in_b;
   tmp3 = in_b - in_a;
   tmp3 = tmp1>>(tmp3 &0x7);
   tmp4 = tmp2<<(tmp1 &0x7);
   return (tmp3 ^ tmp4);
}
Generated code:
20103420 <mashup_16>:
20103420:   33dd 0004             MV    A4,D0
20103424:   33dd 0025             MV    A5,D1
20103428:   3204 18a4             SUB    A6,A5,A4,#0x0
2010342c:   b2e7 b200 3386 0007 20a4 0007
                                MV.S16    A7,#0x7
                             ||    ADD    A8,A5,A4,#0x0
                             ||    AND.U16    A6,#0x7
20103438:   33d2 1d07             AND    A7,A8,A7
2010343c:   b3e4 3204 0108 1085
                                SEXT.16    A8,A8
                             ||    SUB    A4,A4,A5,#0x0
20103444:   33d8 1087             LSL    A4,A4,A7
20103448:   b3d5 7a09 1506        ASR    A5,A8,A6
                             ||    RETD
2010344e:   33e6 10a4             XOR    A4,A5,A4
20103452:   33e4 0084             SEXT.16    A4,A4
20103456:   33e0 0004             MV    D0,A4
```

Note

Since all CPU registers are 32-bits and operations on registers are 32-bits, using 32-bit data variables (for time critical code) in general leads to better performant code.

2.4.6 Coding Style and Impact on Performance

The way the developer writes C code can have an impact on performance. This section illustrates specific example scenarios where this can occur.

- With loops, performance can vary depending on whether the loop counter is a fixed or a variable value. With a fixed value, the compiler has complete knowledge of the loop, and can determine the approach that maximizes performance - whether that means unrolling the loop, software pipelining the loop, and so forth. For example, with matrix multiplication, the performance is significantly better when the matrix row and column sizes are specified in the loops, versus passing them in as function arguments.

- In some cases, merging independent loops into a single loop can speed up performance. The first code block below generates sub-optimal code. The second code block is more optimized.

```
uint8_T Bit_Manipulation_Test_Case(void)
{
uint32_T result;
uint32_T i;
uint8_T valid;
result = 0u;
valid = TC_OK;
i = 0u;
/* Or Test Case */
for(i=0; i<BIT_MANIPULATION_ARRAY_SIZE; i++)
{
    result = (Swc1_Bit_Manipulation.Operand_A[i] | Swc1_Bit_Manipulation.Operand_B[i]);
    if(result != Swc1_Bit_Manipulation.Result_Or[i])
    {
        valid = TC_NOK;
    }
}
/* And Test Case */
for(i=0; i<BIT_MANIPULATION_ARRAY_SIZE; i++)
{
    result = (Swc1_Bit_Manipulation.Operand_A[i] & Swc1_Bit_Manipulation.Operand_B[i]);
    if(result != Swc1_Bit_Manipulation.Result_And[i])
    {
        valid = TC_NOK;
    }
}
/* Xor Test Case */
for(i=0; i<BIT_MANIPULATION_ARRAY_SIZE; i++) {
    result = (Swc1_Bit_Manipulation.Operand_A[i] ^ Swc1_Bit_Manipulation.Operand_B[i]);
    if(result != Swc1_Bit_Manipulation.Result_Xor[i]) {
        valid = TC_NOK;
    }
}
    return valid;
}
```

```
uint8_T Bit_Manipulation_Test_Case(void)
{
uint32_T result_or,result_and,result_xor;
uint32_T i;
uint8_T valid;
result_or = 0u;
result_and = 0u;
result_xor = 0u;
valid = TC_OK;
i = 0u;
/* Or, And, Xor Test Case */
for(i=0; i<BIT_MANIPULATION_ARRAY_SIZE; i++)
{
    result_or = (Swc1_Bit_Manipulation.Operand_A[i] | Swc1_Bit_Manipulation.Operand_B[i]);
    if(result_or != Swc1_Bit_Manipulation.Result_Or[i])
    {
        valid = TC_NOK;
    }
    result_and = (Swc1_Bit_Manipulation.Operand_A[i] & Swc1_Bit_Manipulation.Operand_B[i]);
    if(result_and != Swc1_Bit_Manipulation.Result_And[i])
    {
        valid = TC_NOK;
    }
    result_xor = (Swc1_Bit_Manipulation.Operand_A[i] ^ Swc1_Bit_Manipulation.Operand_B[i]);
    if(result_xor != Swc1_Bit_Manipulation.Result_Xor[i])
    {
        valid = TC_NOK;
    }
}
return valid;
}
```

3 References

1. Texas Instruments, *C29x CPU Reference Guide*
2. Texas Instruments, *F29H85x and F29P58x Real-Time Microcontrollers Data Sheet*
3. Texas Instruments, *F29H85x and F29P58x Real-Time Microcontrollers technical reference manual*
4. Texas Instruments, *Application Software Migration to the C29 CPU user's guide*
5. Texas Instruments, *Implementing Run-Time Safety and Security Protections With the C29x SSU*
6. Texas Instruments, *TI C29x Clang Compiler Tools user's guide*
7. Texas Instruments, *TMS320F2837x, TMS320F2838x, TMS320F28P65x Migration to TMS320F29H85x*

4 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (December 2024) to Revision A (April 2025) **Page**

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

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