Application Note

Optimizing C Code for Size with MSPM0 MCUs: Tips and Tricks

Abstract

When selecting a microcontroller (MCU), the available code space and nonvolatile memory are often critical factors in the decision-making process. To maximize functionality within the constraints of the target device, developers can utilize specific strategies during software development and compilation to achieve code size efficiency. This application note presents various optimization configurations for Code Composer Studio™ (CCS) when working with MSPM0 devices, highlighting compiler settings that can significantly reduce code footprint. Additionally, this document provides coding best practices and techniques for writing more compact, size-optimized code for MSPM0 applications.

Table of Contents

Trademarks

Code Composer Studio™ is a trademark of Texas Instruments.
All trademarks are the property of their respective owners.

Copyright © 2026 Texas Instruments Incorporated

1 Introduction

Strategically choosing compiler settings and writing code with size optimization in mind can make a significant difference when it comes to code size. The code sizes used as an example in this application note are using the code example *i2c_controller_target_dynamic_switching*, which can be built for the MSPM0C1104 MCU, which contains 16kB of Flash and 1kB of SRAM. Code sizes listed were built with CCS version 20.4 with TI Clang version 4.0.4.LTS.

Copyright © 2026 Texas Instruments Incorporated

2 C Compiler Optimization

While C is generally preferred over assembly for readability, C can introduce significant overhead on devices with limited code space. However, with careful compiler optimization and strategic use of features like global variables, C can approach assembly-level efficiency while still benefiting from automated compiler optimizations rather than manual assembly tuning.

2.1 Optimization Settings

In CCS, the optimization level settings can be found in Project > Project Properties… > Build > Arm Compiler > Optimization.

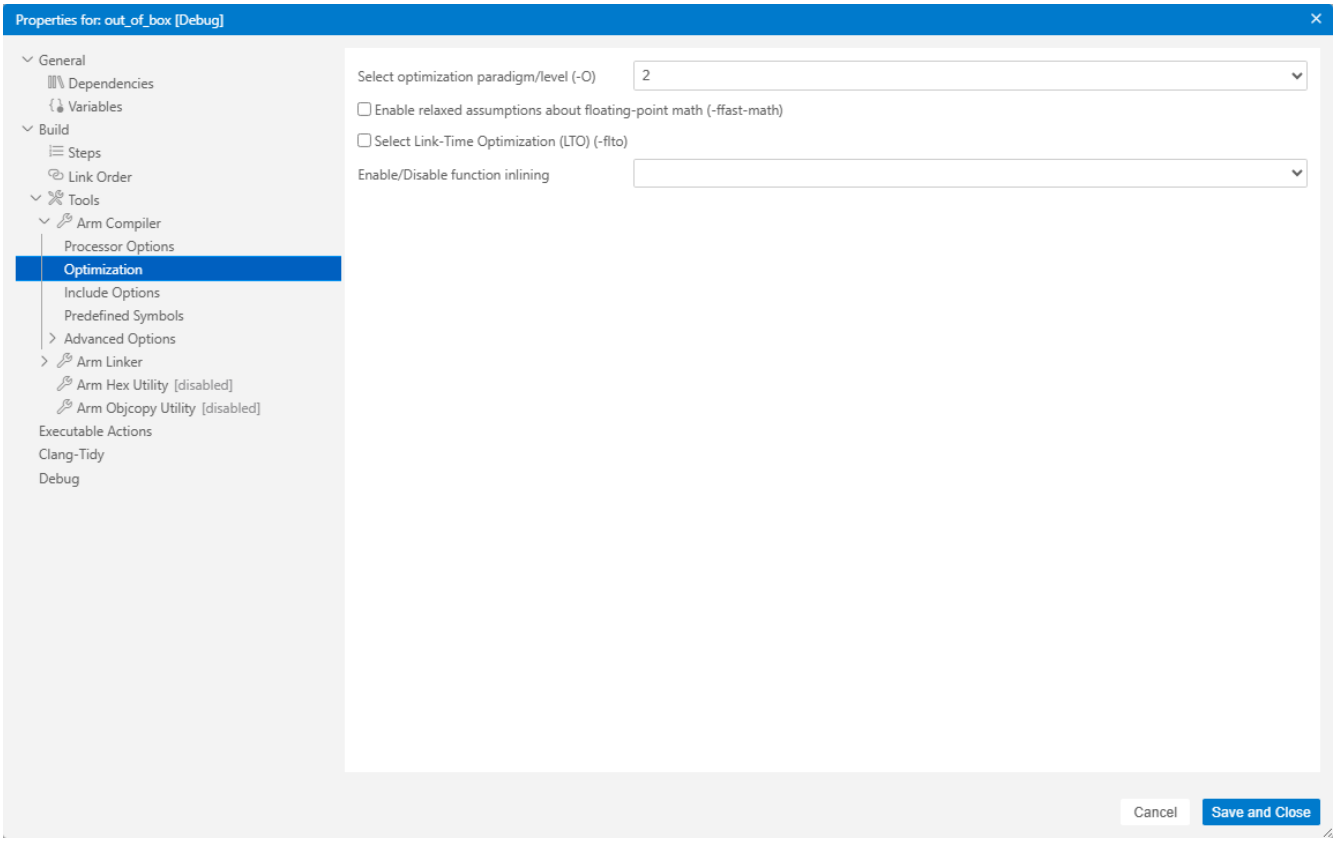


Figure 2-1. Optimization Settings

2.1.1 Optimization Level

The optimization level flag can be configured to favor smaller compiler-generated code size at the expense of performance, or to favor performance at the expense of larger code size. Note that compiler choice directly affects code size due to the distinct optimization algorithms and code generation methods.

Table 2-1. Optimization Level

		Code Optimization Level							
IDE	Keil	O0	O1	O2	O3	Ofast	Os balanced	Oz image size	Og
	CCS	O0	O1	O2	O3	Ofast	Os	Oz	/
	IAR	None	Low	Medium	High	High-speed	High-Balanced	High-Size	/

Copyright © 2026 Texas Instruments Incorporated

Table 2-2. Optimizations Summaries

Optimization Level Option	Optimization Summary	Optimizations Performed
-O0	Performs no optimization. *This option is recommended for maximum debuggability.*	None
-O1, -O	Enables restricted optimizations. Provides a good trade-off between code size and debuggability.	• Control Flow Simplification • Merge contiguous icmps into a memcmp • memcpy/memset/memcmp inlining • Constant Hoisting • Partially inline calls to library functions • Inline for always_inline functions • Global Variable Merging • Merge disjoint stack slots • Loop Strength Reduction • Loop Invariant Code Motion • Common Subexpression Elimination • Dead Argument Elimination • Machine code sinking • Peephole optimization • Tail Predication • Tail Duplication • Load/store optimization • Simple Register Coalescing • Copy Propagation • Conditional Constant Propagation • Called Value Propagation • Control Flow optimization • If-conversion • Thumb2 instruction size reduction • Dead Code Elimination • Loop Vectorization • Printf function specialization • Small memcpy/memset function specialization • Conditionally eliminate dead library calls • Loop Rotation • Loop Unrolling
-O2	Enables most optimizations. Disables some optimizations that can result in larger compiler-generated code sizes.	• Performs all (-O1) optimizations, plus: • Function Integration/Inlining • Instruction speculation • Value Propagation • Jump Threading (non-DFA) • Tail Call Elimination • Merged Load/Store Motion • Global Value Numbering • Memory Dependence Analysis • Dead Store Elimination • Superword-Level Parallelism (SLP) vectorization • Combine redundant instructions • Dead Global Elimination • Global Duplicate Constant Merging • Fast memcpy/memset function specialization • Align loop target boundaries to 16 bytes (Cortex-R4/R5)

Copyright © 2026 Texas Instruments Incorporated

Table 2-2. Optimizations Summaries (continued)

Optimization Level Option	Optimization Summary	Optimizations Performed
-O3	Enables all optimizations available at -O2 plus others that can increase compiler-generated code size. *This option is recommended for optimizing performance, but is likely to increase code size.*	• Performs all (-O2) optimizations tuned for speed, plus: • Replace functions with supported intrinsics • Additional alias analysis and loop optimization • Aggressive Function Inlining • Call-site splitting • Promote 'by reference' arguments to scalars • Combine pattern based expressions
-Og	Enables restricted optimizations while preserving debuggability. Enables all optimizations available at -O1 with the addition of some optimizations from -O2.	• Performs all (-O2) optimizations, but disables the following: • No Loop Vectorization • No function inlining except for always_inline functions • No instruction speculation • No Jump Threading • No Value Propagation • No Tail Call Elimination • No Merged Load/Store Motion • No Global Value Numbering • No Memory Dependence Analysis • No Superword-Level Parallelism (SLP) Vectorization • No Dead Global Elimination • No Global Duplicate Constant Merging
-Os	Enables all optimizations available at -O2 plus additional optimizations design to reduce code size. In comparison to the -Oz option, the -Os option avoids code size optimizations that are likely to harm performance.	• Performs all (-O2) optimizations tuned for code size, plus: • Previously enabled optimizations tuned for code size • Small memcpy/memset function specialization • Minimal memcpy/memset/memcmp inlining • Don't conditionally eliminate dead library calls • Don't align loop target boundaries to 16 bytes (Cortex-R4/R5)
-Oz	Enables all optimizations available at -O2 plus additional optimizations designed to reduce code size. In comparison to the -Os option, the -Oz option performs additional code size optimizations that can harm performance. *This optimization setting is recommended if small code size is a priority.*	• Performs all (-Os) optimizations, plus: • Machine Outlining • No Loop Vectorization • Less aggressive optimizations that impact code size • Don't align loop target boundaries to 16 bytes (Cortex-R4/R5)
-Ofast	This option is deprecated. This option enables all optimizations available at -O3 plus additional aggressive optimizations that are not guaranteed to be in strict compliance with language standards. If you want this behavior, it is recommended that you use the -O3 option in combination with -fast-math.	• Performs all (-O3) optimizations • Additional aggressive optimizations

For examples showing the impact of assembly code at different optimization levels, see the *TI Arm Clang Compiler Tools User's Guide*.

2.1.2 Optnone

The **optnone** function attribute can be applied to a function to instruct the compiler to apply no optimizations in the generation of code for the function. This has the same effect on a single function as the -O0 command-line option has on the entire application.

Copyright © 2026 Texas Instruments Incorporated

Syntax

<return type> *symbol* (<arguments>) __attribute__((**optnone**));

Example

In the following C example, the *park_and_wait* function contains an empty while loop that the compiler optimizes away and removes references to the function if not for the application of the **optnone** attribute to *park_and_wait*:

```
void check_peripheral();
__attribute__((optnone)) void park_and_wait();
void run_a_check_on_peripheral(void) {
check_peripheral();
}
void check_peripheral() {
if (*((unsigned long *)(268612608)) != 286529877) {
park_and_wait();
}
}
__attribute__((optnone)) void park_and_wait() {
while(1) {
;
}
}
```

The compiler-generated assembly for the above code shows that even though the -O2 optimization option is specified, the reference to *park_and_wait* remains intact. If the **optnone** attribute had not been applied to the *park_and_wait* function, the optimizer detects the empty while loop in *park_and_wait* and removes the reference in *check_peripheral*:

```
%> tiarmclang -mcpu=cortex-m4 -O2 -S optnone_function_attr.c
%> cat optnone_function_attr.s
...
.section .text.check_peripheral,"ax",%progbits
.hidden check_peripheral @ -- Begin function check_peripheral
.globl check_peripheral
.p2align 2
.type check_peripheral,%function
.code 16 @ @check_peripheral
.thumb_func
check_peripheral:
movw r0, #46080
movt r0, #4098
ldr r0, [r0]
movw r1, #6485
movt r1, #4372
cmp r0, r1
it eq
bxeq lr
.LBB1_1: @ %if.then
bl park_and_wait
...
```

2.1.3 Link Time Optimization – LTO

LTO enables the compiler to optimize code across multiple source files during the linking phase, rather than optimizing each file individually. LTO can be enabled with the -flto compiler flag.

For Cortex-M0+ processors, enabling LTO achieved 23-25% code size reduction when tested across 191 M0+ Driver Library applications compiled with the -Oz option. The code size reduction percentage can be calculated with the following formula:

% Code Size Reduction = (1 – (LTO code size / non-LTO code size)) × 100

Enabling LTO can also provide significant speedup. For Cortex-M0+ processors, there can be up to a 1.20 Speedup Factor with a 3.8% reduction in time/cycles when tested across 15 EEMBC AutoBench applications compiled with the -O3 option. The following equations can be used to calculate these metrics:

Speedup Factor = (non-LTO cycles / LTO cycles)

% Reduction in Cycles = ((non-LTO cycles - LTO cycles) / non-LTO cycles) × 100

Copyright © 2026 Texas Instruments Incorporated

2.2 Runtime Model Options

In CCS, the Runtime Model Options setting can be found in Project > Project Properties… > Build > Arm Compiler > Advanced Options > Runtime Model Options.

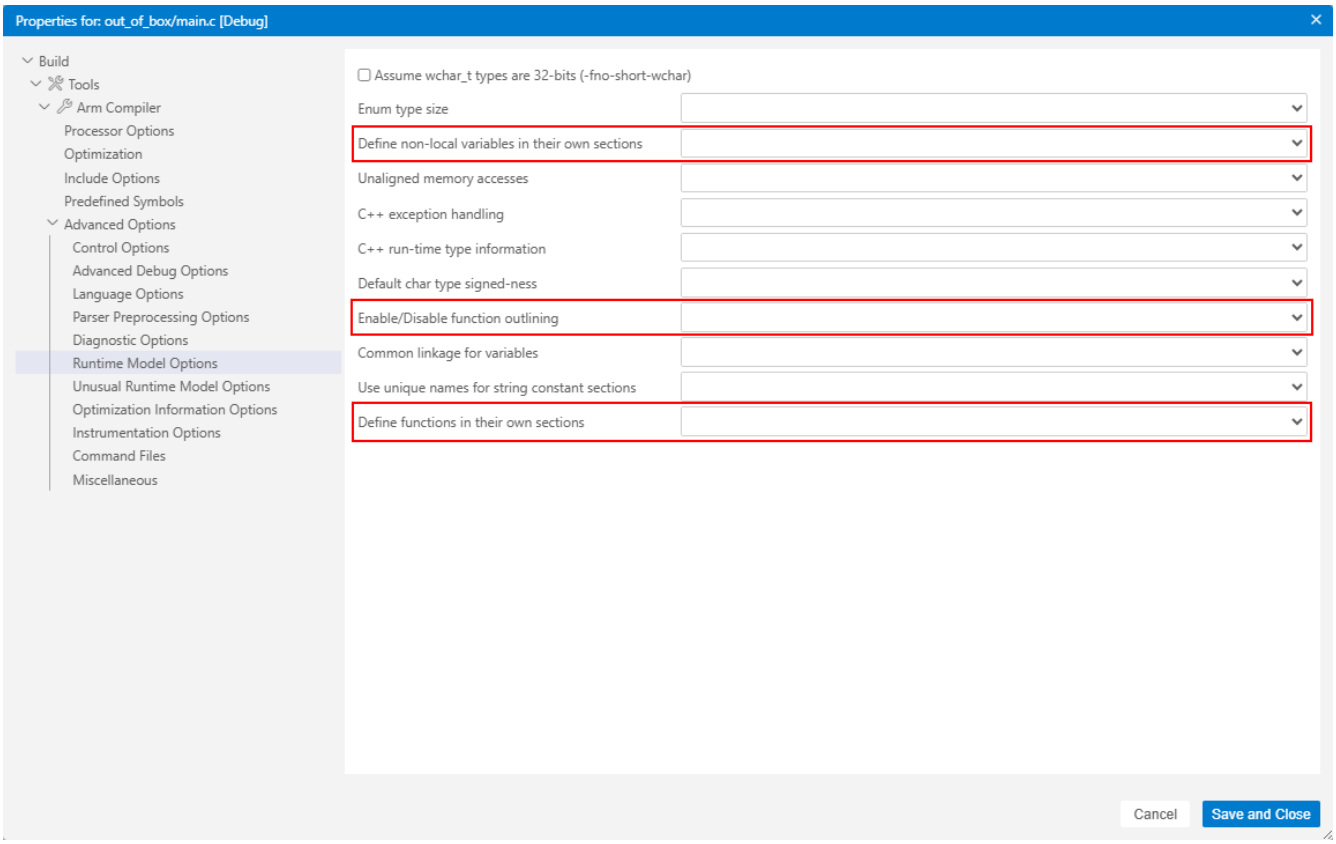


Figure 2-2. Runtime Model Options

2.2.1 Data and Function Sections

The -fdata-sections and -function-sections options are enabled by default. These flags instruct the compiler to generate code for a definition of data and for a function definition in its own section. As a result, these flags enable more granular dead code elimination.

2.2.2 Function Outlining

Function Outlining (aka Machine Outlining) is an optimization that identifies recurring sequences of machine code and replaces each instance with a call to a new function containing those operations. Function Outlining is enabled when the -Oz option is selected.

Table 2-3. Function Outlining

Function Outlining Option	Effect
-moutline	Performs machine outlining within functions; only applied when it is guaranteed that it will reduce the net code size.
-moutline-inter-function	More aggressive function outlining but does not always guarantee an overall code size reduction. Can be specified in combination with the -Oz option to enable inter-function outlining.
-mno-outline	Disables function outlining for a given compilation unit when using the -Oz option.

Copyright © 2026 Texas Instruments Incorporated

3 Coding Techniques

3.1 Use Smallest Possible Types for Variables and Constants

Constants are stored in nonvolatile memory, just like code. Therefore, using the smallest possible type when defining constants helps to reduce wasted code space. For example, if a lookup table contains only values between 0 and 255, using an unsigned 8-bit type when declaring the constant table reduces the space by half compared to a 16-bit int type. This concept also applies to variables stored in Flash using the PERSISTENT keyword. If a variable is stored in Flash instead of SRAM, it uses Flash space that could otherwise be used as code. Therefore, using the smallest type is important in this case as well.

Code can even be written such that smaller numbers are used for the lookup table, provided that the precision is still sufficient for the application. One example is a constant array containing values for timer PWM output. If the timer is sourced from 32768 Hz, but the timer output is only 60Hz, the highest count for the timer period or duty cycle could be only 32768 / 60 = 545. This value cannot be stored in an 8-bit variable. But if the timer source is divided by 4 using the internal clock dividers in the clock module or in the timer, the highest count for duty cycle would now be 8192 / 60 = 136, which is small enough to store in an 8-bit value. Providing that this still provides for enough precision in setting duty cycle, making this simple change halves the size of a const lookup table containing timer count settings. Intelligently choosing how values are stored can make a big difference in the size of arrays and lookup tables.

Beyond choosing the smallest possible data types, you can optimize even further by using bitfields for multiple Boolean flags to pack several flags into a single byte rather than using separate variables. Apply the const qualifier to read-only data to ensure it is stored in Flash memory instead of consuming space in SRAM. Additionally, avoid string literals in your code, as they can quickly consume available memory.

3.2 Avoid Multiply and Divide

Multiply and divide operations take many cycles to perform and require more code to enable these operations. Therefore, finding ways to remove unnecessary multiplication or division from code can be a great way to save on both code space and execution time.

For multiplication or division by powers of 2 (for example, 2, 4, 8, 16, …), bit shifts can be used instead. To do a bit shift in C, use >> to right shift and << to left shift, then the number of bits to shift. This is much more efficient than a multiply or divide because there are assembly instructions and hardware in the CPU for bit shifts.

Another option is to determine if multiplications or divisions can be computed at compile-time rather than runtime. For example, when multiplying two constants with no variable, the compiler can calculate the result during compilation and use the computed value directly, eliminating the need for multiplication code that would otherwise consume both memory and execution time.

If a variable is part of a multiplication, such that it cannot all be calculated ahead, and that variable has a known range of values, a lookup table could be created to contain the possible results.

3.3 Use Lookup Tables Instead of Manual Calculation

If a complex calculation must be repeatedly performed for different values, consider whether a lookup table can be a better alternative – this can be especially true for floating point calculations. If the variable input to the calculation has a known range of values, a lookup table can be created to contain the possible results. Keep in mind that a lookup table does not always build smaller, since the table must also reside in the nonvolatile memory of the device.

3.4 Use Functions Effectively

Functions can make code much easier to read. However, functions can also add overhead in terms of execution speed and code space. If a function is only called once, especially for small functions, the extra overhead is typically not worth it. You can keep a small function for readability but not incur the extra overhead by inlining the function using the inline (or __inline) keyword with the function declaration. This declaration instructs the compiler to insert the contents of the function everywhere the function is called, instead of performing an actual function call.

Copyright © 2026 Texas Instruments Incorporated

However, there are also cases where something needs to be done several times in the code. In this case, a function can make for a more code-size efficient solution, because it allows the same code to be reused in several different places. Writing code intentionally for reusability can make a big difference on code size as well.

3.5 Implement Down Counting Loops

For applications with repetitive actions or looping, choosing the right loop structure can impact code size. Down-counting loops are generally more efficient than up-counting loops because they take advantage of the processor's built-in zero flag. When a loop counts down to zero, the decrement instruction automatically sets the zero flag, allowing the loop termination condition to be checked without additional comparison instructions. In contrast, up-counting loops require explicit comparison instructions and must load the target value for each iteration.

3.6 Use the __even_in_range() Intrinsic

The __even_in_range(x,NUM) intrinsic provides a hint to the compiler for switch statements that the value x will always be an even value in the range 0 to NUM inclusive. This allows the compiler to make more assumptions about the value of x and optimize further than normal. This is typically used for interrupt service routines (ISRs) because interrupt vectors always have a fixed range of values and are always even.

__even_in_range() is not limited to ISRs, however. The intrinsic can also be used on other switch statements in code, as long as the code maintains that the control variable for the switch is always even and has an upper limit on its value. An example is a byte counter variable that increments by 2 instead of by 1, and has a maximum value controlled by code so that it rolls over to 0 or 2.

3.7 volatile Keyword

The **volatile** keyword is part of the C standard. The tiarmclang compiler supports this keyword in all language modes.

The **volatile** keyword indicates to the compiler that there is something about how the variable is accessed that requires that the compiler not use overly-clever optimization on expressions involving that variable. For example, the variable can also be accessed by an external program, an interrupt, another thread, or a peripheral device.

The compiler eliminates redundant memory accesses whenever possible, using data flow analysis to figure out when it is legal. However, some memory accesses can be special in some way that the compiler cannot see, and in such cases you should use the **volatile** keyword to prevent the compiler from optimizing away something important. The compiler does not optimize out any accesses to variables declared volatile. The number of volatile reads and writes are exactly as they appear in the C/C++ code, no more and no less and in the same order.

Any variable that might be modified by something external to the obvious control flow of the program (such as an interrupt service routine) must be declared **volatile**. This tells the compiler that an interrupt function might modify the value at any time, so the compiler can not perform optimizations which change the number or order of accesses of that variable. This is the primary purpose of the volatile keyword.

Copyright © 2026 Texas Instruments Incorporated

4 Example

Upon building the example, i2c_controller_target_dynamic_switching, with optimization level -O2, the memory allocation results in ~2.5kB of Flash and 117B of SRAM used.

Figure 4-1. Memory Allocation

4.1 Optimization Level

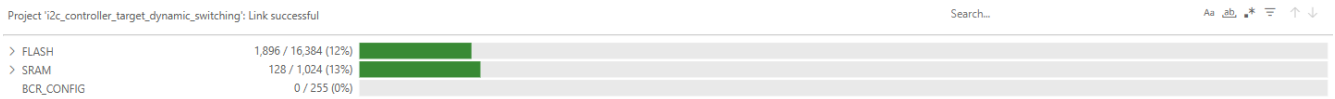
Table 4-1 shows the code size of the example with different optimization levels.

Table 4-1. Code Size

Optimization Level	Code Size
-O0	4,112 B
-O1	2,208 B
-O2	2,464 B
-O3	2,456 B
-Og	2,208 B
-Os	2,160 B
-Oz	2,128 B

4.2 LTO

When LTO is enabled with the -Oz optimization level, the code size is reduced to 1,896B, while SRAM usage slightly increases to 128B.



4.3 Runtime Model Options

The following tests were performed with optimization level -O2.

Table 4-2. Runtime Model Options

LTO	Outlining	Data/Function Sections	Code Size Flash / SRAM
No	-moutline	Default (enabled)	2,456 B / 105 B
No	-moutline-inter-function	Default	2,456 B / 105 B
No	-mno-outline	Default	2,464 B / 105 B
No	-moutline	-fno-data-sections -fno-function-sections	2,520 B / 105 B
Yes	-moutline	Default	2,328 B / 105 B

Copyright © 2026 Texas Instruments Incorporated

5 Summary

Testing different approaches is often the best way to find optimal coding solutions. Leveraging version control systems allows you to create separate branches or copies of your project to try different optimization methods. You can then compare the results to see which works best.

Copyright © 2026 Texas Instruments Incorporated

6 References

1. Texas Instruments, *TI Arm Clang Compiler Tools User's Guide - 5.0.0.STS*
2. Texas Instruments *Optimizing C Code for Size With MSP430™ MCUs: Tips and Tricks* application report

Copyright © 2026 Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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 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, TI’s General Quality Guidelines, 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. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

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

Copyright © 2026, Texas Instruments Incorporated

Last updated 10/2025