



ABSTRACT

This guide is intended to provide information on issues to focus on when migrating application software from the C28 CPU to the C29 CPU, and from the CLA to the C29 CPU.

Table of Contents

1 Introduction	2
2 C28 to C29 CPU Migration	2
2.1 Use Cases.....	2
2.2 Key Differences.....	2
2.3 Source Code Migration.....	2
2.4 Toolchain Migration.....	5
3 CLA to C29 CPU Migration	6
3.1 Use Cases.....	6
3.2 Key Differences.....	6
3.3 Source Code Migration.....	7
3.4 Toolchain Migration.....	10
4 References	10

List of Tables

Table 2-1. C28 vs C29 CPU Programming Models.....	2
Table 2-2. Data Type Key Differences.....	3
Table 3-1. CLA vs C29 Programming Model Differences.....	6
Table 3-2. Data Type Key Differences.....	8

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

This guide is intended to provide information on issues to focus on when migrating application software from the C28 CPU to the C29 CPU, and from the CLA to the C29 CPU.

2 C28 to C29 CPU Migration

2.1 Use Cases

This is applicable for users migrating from the F28x platform (C28 + CLA) to the F29x platform (C29).

2.2 Key Differences

The key differences between the C28 and C29 programming models are summarized in [Table 2-1](#)

Table 2-1. C28 vs C29 CPU Programming Models

Item	C28 CPU	C29 CPU
Instruction issue per clock cycle	Single	VLIW - Multiple (up to 8)
Pipeline stages	8	9
Memory map	Fragmented	Contiguous
8b byte addressability	No	Yes
Registers	8 x 64-bit floating point (with double-precision FPU64) 8 x 32-bit floating point (with single-precision FPU) 3 x 32-bit fixed point (ACC, P, XT) 8 x 32-bit addressing, SP, DP	16 x 64-bit floating point (pairs of 32-bit floating point registers) 8 x 64-bit fixed point (pairs of 32-bit fixed point registers) 16 x 32-bit addressing
Stack reach	16-bit Stack Pointer (SP)	32-bit Stack Pointer (A15)
Instruction width(s)	16-bit, 32-bit	16-bit, 32-bit, 48-bit
Bus widths	1 x 32-bit Data Read (32-bit address) 1 x 32-bit Data Write (32-bit address) 1 x 32-bit Program Read (22b address)	2 x 64-bit Data Read (32-bit address) 1 x 64-bit Data Write (32-bit address) 1 x 128-bit Program Read(32-bit address)

2.3 Source Code Migration

This section describes the key issues when migrating source code from the C28 to the C29 CPU. It also discusses tooling available to aid users with migration.

2.3.1 C/C++ Source Code

One of the key concerns when porting platforms is for the order of memory accesses to be maintained, which would be the case since the C28 and C29 CPU pipelines have the same sequence of steps. However, there are some key differences that impact C/C++ source code that are discussed below. The C28 compiler supports some C/C++ extensions that the C29 compiler does not.

2.3.1.1 Pragmas and Attributes

- The use of target-agnostic pragmas and attributes supported by clang are very likely to be portable, whereas a number of C28-specific pragmas and attributes will not be. See the compiler user guide for details.
- The C29 compiler has a tool called c29clang-tidy which checks for use of C28 pragmas and suggest alternatives if available. This is discussed [here](#) (under c29migration-c28-pragmas)
- The C29 Compiler user guide also discusses pragmas and attributes [here](#) and [here](#)

2.3.1.2 Macros

- Several C28-specific predefined macros are not supported in the C29 Compiler.
- The C29 Compiler user guide discusses macros and their migration [here](#) and [here](#).

2.3.1.3 Ininsics

- C28 Ininsics are builtin functions that map to one or more C28 assembly instructions. Some C28 intrinsics have functionally equivalent C29 intrinsics, but many do not.

- The C29 compiler has a tool called c29clang-tidy which checks for use of C28 intrinsics and suggest alternatives if available, which is discussed [here](#) (under c29migration-c28-builtins)
- The C29 Compiler user's guide also discusses intrinsics [here](#) and [here](#)

Note

The C28 compiler maps several RTS library calls to TMU instructions when the compiler settings are appropriately set (--fp_mode=relaxed). This optimization is not yet implemented in the C29 compiler. It is recommended that, for now, users use C29 intrinsics corresponding to TMU instructions to efficiently implement such operations.

2.3.1.4 Inline assembly

- Inline assembly - if user C code contains inline assembly, users will need to manually update this, since the C28 and C29 instruction sets are different.

2.3.1.5 Keywords

- Keywords - The C28 compiler supports all of the standard C89 keywords, including const, volatile, and register. It supports all of the standard C99 keywords, including inline and restrict. It supports all of the standard C11 keywords. It also supports TI extension keywords __interrupt, __cregister, and __asm. Some of these (for example, __interrupt, __cregister) do not port to C29.

There is no support for keyword checking in the C29 compiler's c29clang-tidy tool. If a keyword is not supported, the C29 compiler will generate an error. Users need to manually address such keywords. If applicable, they can be #if defined to nothing, as shown below.

```
#ifndef __c29__
# define CREGISTER
#elif defined(__TMS320C2000__)
# define CREGISTER __cregister
#endif
```

2.3.1.6 Data Type Differences

Data type differences - the key differences in this context are summarized and highlighted in [Table 2-2](#).

Table 2-2. Data Type Key Differences

Type	C28	CLA	C29	ARM
char	16	16	8	8
short			16	
int	16	32	32	32
long			32	
long long (COFF)	64	32	N/A	64
long long (EABI)			64	
float			32	
double (COFF)	32	32	N/A	64
double (EABI)			64	
long double (COFF)	64	32	N/A	64
long double (EABI)			64	
Pointers	32	16	32	32

The user needs to pay careful attention to data types:

1. int (C28 16-bit vs C29 32-bit) - If the user's code has occurrences of "int", the following is recommended for migration:
 - a. Change all occurrences of "int" in user code to a fixed-width type int16_t. This ensures nothing in the code breaks, and original data widths are retained.
 - i. The C29 compiler has a tool called c29clang-tidy which checks for occurrences of "int" and "unsigned int", which is discussed [here](#) (under c29migration-c28-int-decls)
 - b. Ideal scenario - user code does not contain any occurrences of "int", instead only contains portable fixed-width types like int16_t, int32_t. In this case, the code is portable without any changes.
2. char (C28 16-bit vs C29 8-bit) - If the user's code has occurrences of "char" or "int8_t" or "uint8_t":
 - a. At the outset, changing all occurrences of "char" to "int16_t" may seem like the migration approach, since it is retaining the bit-width present in the user's original code. However, this can lead to unexpected build-time issues, or worse, hard to detect run-time failures. This is because it is not legal for pointers of different types to alias. Char, however, is special and may alias with other pointer types. In other words, any object can be accessed through a pointer to char. So if you now change char to int16_t, you will violate this "special privilege" that char alone has.
 - b. Therefore, the recommended approach to migration is to make no changes to the code, or preferably, change all occurrences of "char" in user code to "int8_t". Even this approach has some problematic scenarios:
 - i. If user code contains "char" occurrences but really has 16-bit dependency, which needs to be changed to int16_t. For example, assigned data that is out of 8-bit range and needs 16-bit range. The C29 compiler has a tool called c29clang-tidy that checks for operations on "char" typed expressions that would be out of range for an 8-bit type, which is discussed [here](#) (under c29migration-c28-char-range)
 - ii. If user code contains pointers to "char" and associated offsets, or arithmetic on such pointers. The C29 compiler has a tool called c29clang-tidy that checks for pointer arithmetic on char/int-based pointers, whose bit stride changes between C28 to C29, which is discussed [here](#) (under c29migration-c28-types and c29migration-c28-suspicious-dereference).

Note

Note that Pointer aliasing is not an issue with changing "char" to "int8_t".

3. Differences in sizeof() due to the data type differences - This is summarized in the table, and leads to differences in behavior when using standard library function calls:

	sizeof(char)	sizeof(short)	sizeof(int)	sizeof(long)
C28	1	1	1	2
C29	1	2	4	4

- a. If a hardcoded size is used, the C29 compiler has a tool called c29clang-tidy which checks for library calls without a sizeof() expression, which is discussed [here](#) (under c29migration-c28-stdlib).
- b. Functions which work on values byte-by-byte, like memset, do not port from C28 to C29. If memset is used with sizeof(int), sizeof(int16_t), sizeof(char), or sizeof(int8_t), the behavior is different between C28 and C29. Consider the following example with memset. Some of the functions that are affected are memset, memccpy, memchr, strncmp.

```
memset(buf,5,2 * sizeof(char));
Byte address offset at buf: 0 8 16 24 32 40 48 56 64
C28:           5   5
C29:           5  5
```

```
memset(buf,5,2 * sizeof(short));
Byte address offset at buf: 0 8 16 24 32 40 48 56 64
C28:                        5    5
C29:                        5 5 5 5
```

```
memset(buf,5,2 * sizeof(int));
Byte address offset at buf: 0 8 16 24 32 40 48 56 64
C28:                        5    5
C29:                        5 5 5 5 5 5
```

Differences in behavior occur with other functions like memcpy as well, even if they don't operate byte wise. Consider the following example with memcpy.

```
memcpy(dst,src,4 * sizeof(char));
Byte address offset at dst: 0 8 16 24 32 40 48 56 64
C28:                        1 2 3 4 5 6 7 8
C29:                        1 2 3 4
```

Note how with memcpy, in order to obtain the same behavior on both C28 and C29, CHAR_BIT (defined in limits.h) can be used.

```
#if(CHAR_BIT == 16)
memcpy(dst,src,4 * sizeof(char));
#endif
#if(CHAR_BIT == 8)
memcpy(dst,src,4 * sizeof(char) * 2);
#endif
Byte address offset at dst: 0 8 16 24 32 40 48 56 64
C28:                        1 2 3 4 5 6 7 8
C29:                        1 2 3 4 5 6 7 8
```

Note

The portability of fixed-width types is possible with the use of C headerstdint.h.

2.3.1.7 Tooling support for Migration

As mentioned a number of times above, the C29 Compiler has a utility called c29clang-tidy that aids in C/C++ source code migration from C28 to C29. It is discussed [here](#).

2.3.2 Assembly Language Source Code

Any user code written in C28 assembly language does not port over to C29, since the instruction sets are different. Performance entitlement with the C29 VLIW architecture is expected with the C compiler, therefore users are strongly encouraged not to write C29 assembly code. For users who have written C28 assembler constructs and are looking for equivalent constructs in the C29 tool chain, refer [here](#).

For more information, refer to this topic in the *C29 Clang Compiler Tools User's Guide* [here](#).

2.4 Toolchain Migration

This section describes high-level differences between the C28 and C29 toolchains, and points to relevant documentation.

2.4.1 Compiler

The C28 compiler and C29 compiler use different underlying infrastructures and completely different source bases.

- The C28 compiler is TI proprietary, whereas the C29 compiler is LLVM-clang-based. Compiler options are completely different, and need to be changed. Many concepts behind the meaning of options have changed as well, such as for optimization. The *C29 Clang Compiler Tools User's Guide* is available [here](#). It has a detailed section on migration, available [here](#).
- The C29 compiler supports only EABI output formats, whereas, the C28 compiler supports both COFF and EABI. A user migrating from C28-COFF to C29 should first migrate from COFF to EABI, and can refer to documentation available [here](#).

Note

The C29 CPU is a VLIW architecture and allows for significantly higher level of parallelism compared to C28 and CLA. However, for the C29 compiler to fully take advantage of these parallel functional units in the C29, users have to use higher levels of optimization (e.g. -o2, -o3).

2.4.2 Linker

The tool chain's linker remains the TI linker, not the LLVM linker. This means that, in general, the syntax of linker command files does not change. However, the way that a linker command file is specified during compilation has changed. [This page](#) contains detailed information on Linker flags within LLVM. Information on migrating linker command files for use with the C29 compiler is available [here](#).

Before:

```
c12000 a.c -z -l1nk.cmd
```

After:

```
c29lang a.c -w1,-l1nk.cmd
```

- There are some features implemented for the C28 compiler that interacted with the TI linker and involved unique specification in the linker command file. Where these features are applicable to the C29, they will be addressed in a future version of the compiler e.g. Live Firmware Update (LFU), which is supported by the C28 Compiler.

2.4.3 CCS Project Migration

C29 based devices will only be supported on the CCS Theia IDE and not the CCS IDE, whereas C28 based devices are supported on the CCS IDE. Information on migrating from C28 compiler options to C29 compiler options is available [here](#). For additional information on project migration, refer to the docs folder of the C29 SDK.

3 CLA to C29 CPU Migration

3.1 Use Cases

This is applicable to users migrating from a C28+CLA device to a C29 device. In this context, the migration would be from the CLA to the C29.

Note

This section is based on existing F29x devices which do not contain a CLA.

3.2 Key Differences

The key differences between the CLA and C29 Programming models are summarized in [Table 3-1](#).

Table 3-1. CLA vs C29 Programming Model Differences

Item	CLA	C29
Instruction issue per clock cycle	Single	VLIW - Multiple (up to 8)
Pipeline stages	8	9
Memory access	LSRAM only	RAM and Flash
8b byte addressability	No	Yes
Registers	4 x 32-bit floating point 2x 16-bit Auxiliary	16 x 64-bit floating point (pairs of 32-bit floating point registers) 8 x 64-bit fixed point (pairs of 32-bit fixed point registers) 16 x 32-bit addressing
Stack reach	16-bit Stack Pointer (SP)	32-bit Stack Pointer (A15)

Table 3-1. CLA vs C29 Programming Model Differences (continued)

Item	CLA	C29
Instruction width(s)	32-bit	16-bit, 32-bit, 48-bit
Bus widths	1 x 32-bit Data Read (32-bit address) 1 x 32-bit Data Write (32-bit address) 1 x 32-bit Program Read (16-bit address)	2 x 64-bit Data Read (32-bit address) 1 x 64-bit Data Write (32-bit address) 1 x 128-bit Program Read (32-bit address)

3.3 Source Code Migration

This section discusses source code migration from the CLA to C29.

3.3.1 C/C++ Source Code

There are key issues that can be better understood by considering

- Existing C28+CLA development, and how system code is developed to run on both the C28 CPU and the CLA.
 - Whether the migration is to the same C29 CPU to which C28 code is being migrated, or a different C29 CPU. Each has its benefits and challenges.
1. When moving from CLA to C29:
 - a. .cla files would need to be converted to .c files
 - b. CLA tasks would need to be mapped to interrupts on C29 CPU.
 - i. If possible, use RTINT instead of INT for hardware context-save and superior performance
 - ii. Since CLA tasks run to completion and cannot be preempted, to achieve similar functionality, the interrupts can be assigned into a group. Interrupts in a given group do not preempt or nest within other interrupts in the group. For more details, see the *PIPE* chapter of the F29x device-specific Technical Reference Manual.
 - iii. The CLA background task is interruptible, so if it is enabled, it should be in a lower nesting group than the group corresponding to the rest of the CLA tasks. Another option the user can consider is moving the background task functionality into the idle loop in main().
 2. Code and data placement in LSRAM constraints would be removed. With the CLA, code and data both have to reside in LSRAM. On the C29 device, CLA code can be migrated to run on the LPAX RAM, and data can reside in the LDAX RAM.
 3. The CLA compiler has C language standard restrictions. These are mentioned in the CLA compiler description in the [TMS320C28x Optimizing C/C++ Compiler v22.6.0.LTS User's Guide](#). For example, defining and initializing global/static data is not supported. The use of function pointers is not supported. These restrictions are lifted when moving to the C29.
 4. The CLA compiler supports a subset of C28 pragmas and attributes. These need to be addressed when porting from the CLA to the C29.
 5. Keywords - The CLA compiler supports all but 2 of the keywords (far and ioport) supported by the C28 compiler.
 6. For more information, see [section](#) in the compiler guide that discusses migrating CLA source code.

3.3.1.1 Data Type Differences

1. Data type differences - are summarized and highlighted in [Table 3-2](#): Red for CLA-C29 differences and Blue for CLA-C28 differences.
 - a. Shared data - Data type differences between the C28 and CLA have an impact when data is shared between the C28 and CLA (shared structures). The recommended approach here is to use padding through a union to solve the pointer or integer size difference as described in the CLA software development [guide](#). For example, unions are used in the C2000Ware DigitalPower SDK solutions for enums (int based) shared between C28 and CLA. These enums define which Lab, board status, and so forth are being used. Since C29 devices do not contain a CLA, there is no issue of shared structures, so the above data type differences are not of concern from that perspective.

- b. However, data type differences between the C29 and CLA are relevant when migrating code from CLA to the C29. The issues and how to address them are similar to the items laid out in the C28-C29 data type differences (see [Section 2.3.1.6](#)).
 - i. Pointer size differences exist between the CLA (16-bit) and C29 (32-bit).
 - ii. "Char" differences, as described in the C28 to C29 source code migration section. The C29 compiler's c29clang-tidy tool's checkers on char range (c29migration-c28-char-range) and char pointer arithmetic (c29migration-c28-types) are useful here as well.

Table 3-2. Data Type Key Differences

Type	C28	CLA	C29	ARM
char	16	16	8	8
short	16			
int	16	32	32	32
long	32			
long long (COFF)	64	32	N/A	64
long long (EABI)	64			
float	32			
double (COFF)	32	32	N/A	64
double (EABI)	64			
long double (COFF)	64	32	N/A	64
long double (EABI)	64			
Pointers	32	16	32	32

3.3.1.2 Migrating CLAmath.h Functions and Intrinsics

1. The CLA compiler does not support math.h. A separate file CLAmath.h is used, which contains external references to functions/variables in the C2000Ware CLAmath library. These are hand optimized CLA assembly routines for specific operations (trig, div, sqrt, isqrt, exp, log). It also contains function redefinitions/mappings for code portability from C28 to CLA, such as:
 - a. Mapping specific math.h functions and TMU intrinsics to CLA math library functions mentioned above. For example, if C28 code contains `__cos` (corresponding to the TMU instruction) or `cosf`, it will migrate without any updates to CLA because CLAmath.h maps `__cos` and `cosf` to `CLAcos` (in the CLAmath library) or `CLAcos_inline` (in CLAmath.h).
 - i. So if user CLA code contains `__cos`, migrating to C29 will be similar to migrating C28 intrinsics to C29, as discussed in the C28-C29 source code migration section.
 1. The C29 compiler has a tool called c29clang-tidy which checks for use of C28 intrinsics and suggest alternatives if available, which is discussed [here](#) (under c29migration-c28-builtins). It is applicable here as well.
 - ii. If it contains `cosf`, migrating to C29 does not require any changes.
 - iii. However, if it contains `CLAcos` or `CLAcos_inline`, then the user needs to change all these calls to `cosf`. **A mapping header file to assist in migration of CLAmath functions is planned.**
 - b. Mapping specific math.h functions and C28+FPU intrinsics to CLA intrinsics. For example, if C28 code contains `__fmax` (corresponding to the FPU instruction) or `fmaxf`, it will migrate without any updates to CLA because CLAmath.h maps `__fmax` and `fmaxf` to `__mmaxf32` (the CLA intrinsic).
 - i. So if user CLA code contains `__fmax`, migrating to C29 will be similar to migrating C28 intrinsics to C29, as discussed in the C28-C29 source code migration section.
 1. The C29 compiler has a tool called c29clang-tidy which checks for use of C28 intrinsics and suggest alternatives if available, which is discussed [here](#) (under c29migration-c28-builtins). It is applicable here as well.
 - ii. If it contains `fmaxf`, migrating to C29 does not require any changes.
 - iii. However, if it contains `__mmaxf32`, then the user needs to change all these calls to `fmaxf`. **c29clang-tidy extension to check for CLA intrinsics is planned.**

- c. If user code contains CLA intrinsics that do not map to a corresponding C28 intrinsic (for example, `__mgeq`, `__mgequ`, `__mgt`, `__mgtu`, `__mleq`, `__mlequ`, `__mlt`, `__mltu`, `__mdebugstop`), user code needs to be manually updated to fix them. **c29clang-tidy extension to check for CLA intrinsics is planned.**

3.3.1.3 Migrating C28 and CLA to the Same C29 CPU

1. In the C28+CLA implementation, CLA tasks could be triggered by hardware or software, and on task completion, an interrupt may be sent to the C28.
2. To implement this in this scenario, for software task triggers, a software interrupt would be triggered through PIPE to run the desired ISR.
3. For hardware task triggers, the C29 CPU's PIPE would be setup to trigger from the desired peripheral.
4. For task completion interrupts, a software interrupt is triggered through PIPE (user code needs to write to PIPE registers) to run the desired ISR.
5. If a background task was present in the C28+CLA implementation, it can be easily implemented in the C29 CPU as a background loop (idle loop).
6. One key challenge in this scenario is interrupt priority assignment, since both C28 ISRs and CLA tasks are now mapped to C29 ISRs and need interrupt priority assignment. Whereas, on the C28+CLA, they run independently on independent cores, here they run on the same C29 CPU. CLA tasks run to completion without being preempted by other tasks, and as mentioned above, this can be achieved by grouping them into the same C29 PIPE interrupt group. However, there is no way to ensure C28 ISRs do not preempt CLA tasks. Likewise, there is no way to ensure CLA tasks do not preempt C28 ISRs. Therefore, in this case, the user must necessarily perform analysis of C28 ISRs + CLA tasks as a whole and determine the interrupt priorities that suit the application.
7. Similarly, if a C28 ISR was software triggering a CLA task, with them now resident on the same C29 CPU, this presents a problem of relative interrupt priorities of each. If the CLA task is made higher priority, it can preempt the C28 ISR and run and deliver the expected functionality from a CLA side. However, this means the C28 ISR is halted and this may not be expected functionality from the C28 perspective.
8. Also, if the same exact event (for example, peripheral) triggers a C28 ISR as well as a CLA task, now with them running on the same C29 CPU, the user could merge them into a single ISR.
9. CLA registers offer a lot of functionality to users, like being able to know which task is running, and being able to stop a task by writing to a specific bit in a register. When migrating, suitable source code updates may be needed given the absence of these registers and corresponding functionality.

Note

Not all corner case scenarios may have been identified here.

3.3.1.4 Migrating C28 and CLA to Different C29 CPUs

1. A different CCS project is needed.
2. In the C28+CLA implementation, CLA tasks could be triggered by hardware or software, and on task completion, an interrupt may be sent to the C28.
3. To implement this in this scenario, you would need IPC between two C29 CPUs. IPC, however, has a maximum of four interrupts in either direction, whereas, there could be eight CLA tasks. Hence, it would make sense to use one IPC interrupt in either direction, along with a command specifier to indicate the task that needs to be run or the task that completed. This requires additional code that parses the IPC interrupt and triggers the ISR.
4. So, for software task triggers, you would map them to an IPC interrupt, then the receiving C29 CPU would run the IPC ISR, and trigger a software interrupt through its PIPE (by writing to PIPE registers) to run the desired ISR.
5. For hardware task triggers, the C29 CPU's PIPE would be setup to trigger from the desired peripheral.
6. For task completion interrupts, you would map it to an IPC interrupt, then the receiving C29 CPU would run the IPC ISR, and trigger a software interrupt through its PIPE to run the desired ISR.
7. If a background task was present in the C28+CLA implementation, it can be easily implemented in the C29 CPU as a background loop (idle loop).
8. CLA registers offer a lot of functionality to users, like being able to know which task is running, and being able to stop a task by writing to a specific bit in a register. When migrating, suitable source code updates may be needed given the absence of these registers and corresponding functionality.

3.3.2 Assembly Language Source Code

Any user code written in CLA assembly language does not port over to C29, since the instruction sets are different. Performance entitlement with the C29 VLIW architecture is expected with the C compiler, therefore users are strongly encouraged to write equivalent C code and take advantage of the C29 architecture's parallelism as well as the performance entitlement offered by the compiler.

3.4 Toolchain Migration

Since C29 devices do not contain a CLA, the C29 compiler does not need to implement a corresponding CLA compiler. The user's C28+CLA project may contain CLA-specific compiler options that is not relevant in the C29 toolchain.

The existing memory model for the CLA requires specific sections for specific types of data (.bss_cla, .const_cla, .scratchpad, no heap, and so forth). The user's C28+CLA project's linker command file contains these sections and mappings, which are not relevant in the C29 toolchain. The user needs to modify these.

C29-based devices will only be supported on CCS Theia, whereas C28-based devices are supported on the CCS IDE. For additional information on project migration, refer to the docs folder of the C29 SDK.

4 References

1. C29 Compiler User [Guide](#)
2. Texas Instruments: [TMS320C28x Optimizing C/C++ Compiler v22.6.0.LTS User's Guide](#)
3. Texas Instruments: [TMS320C28x CPU and Instruction Set Reference](#)
4. Texas Instruments: [F29H85x and F29P58x Real-Time Microcontrollers Technical Reference Manual \(SPRUJ79\)](#)

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](https://www.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
Copyright © 2024, Texas Instruments Incorporated