*Application Note*

# *Bad Block Management for Serial NAND Flash on Sitara Devices*

*Aryamaan Chaurasia, Vaibhav Kumar, Hong Guan, Benoit Parrot*

**ABSTRACT**

This Application Note describes the Bad Block Management implementation for Serial NAND Flash memory on TI's Sitara™ devices and explains the architectural approach and mechanisms for detecting, tracking, and avoiding bad blocks across three environments: ROM Bootloader, TI™ MCU+ SDK, and TI™ Processor SDK. The target audience includes embedded systems engineers and developers working with Serial OSPI NAND Flash on Sitara™ processors.

## Table of Contents



## Trademarks

# 1 Introduction

NAND Flash memory devices inherently contain defective blocks that cannot reliably store data. Since the manufacturing process cannot economically guarantee 100% defect-free yield, the high-density nature of NAND Flash makes occasional defects inevitable during fabrication. Manufacturers accept this reality and ship devices with a small percentage of pre-marked bad blocks, typically 0–2% of the total capacity.

These bad blocks may originate from the manufacturing process as factory bad blocks or develop during the device's operational lifetime which are runtime bad blocks. A robust bad block management system is essential for reliable Serial OSPI NAND flash operation, ensuring data integrity by detecting, tracking, and avoiding defective blocks.

This Application Note describes the architectural approach and key mechanisms for implementing Bad Block Management in OSPI NAND Flash systems, providing guidance for embedded applications.

# 2 Bad Block Management

Bad Block Management is a system-level mechanism that identifies, tracks, and handles defective blocks in Serial NAND Flash memory devices. A bad block is a block that contains one or more invalid bits and cannot reliably store and retrieve data.

A bad block marker is a byte pattern, typically in the spare/Out-Of-Band area of the NAND Flash that identifies defective memory blocks that should not be used. Bad Block Management is implemented by marking good blocks with 0xFF, while marking bad blocks with a non-0xFF value.

- Bad Block Management consists of several key functions:
  - Detection: Identifying which blocks in the Flash device are defective.
  - Tracking: Maintaining a list or table of bad block locations.
  - Avoidance: Skipping bad blocks during Flash erase, write, and read operations.
  - Marking: Marking newly discovered bad blocks for future avoidance.

The diagram depicted below showcases the memory architecture for a Winbond W35N01JW Serial NAND Flash that is mounted by default on the AM62Ax device.

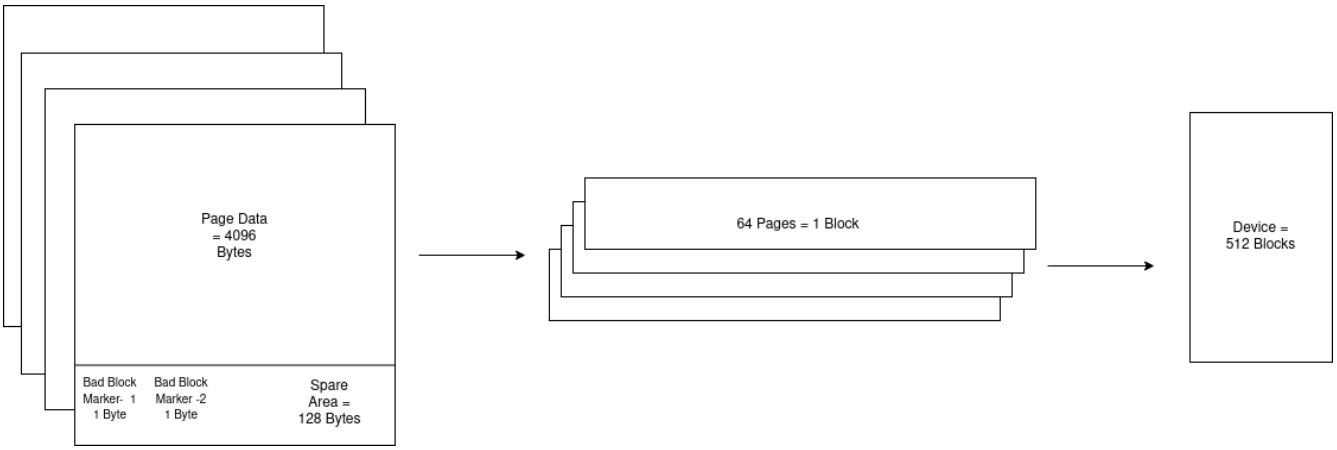


**Figure 2-1. Flash Memory Architecture for W35N01JW Winbond Flash**

## 2.1 Block Failure

NAND Flash blocks can fail for various reasons, occurring either during manufacturing or during the device's operational lifetime. The following factors contribute to block failure:

- Manufacturing Defects: Particle Contamination, Lithography Errors, Doping Irregularities, Material Defects.
- Program/Erase Failure: Blocks that cannot be fully erased to the 0xFF state become unusable.
- Physical Damage: Electrical Overstress, Temperature Extremes, Mechanical Stress, Radiation.

## 2.2 Feasibility of Reversing Bad Blocks

Bad blocks cannot be reversed or repaired once they develop. The physical mechanisms causing block failure are permanent and irreversible. Bad Blocks represent hard failures that persist across power cycles and cannot be cleared.

# 3 Bad Block Management in ROM Bootloader

The ROM Bootloader (RBL) initiates the boot process by loading the Secondary Bootloader (SBL) from Flash memory, starting at location 0x00. To ensure data integrity, the RBL first checks the bad block marker of the initial block. If the block is marked as bad, the RBL iteratively checks subsequent blocks, examining their bad block markers until it identifies a good block. Once a valid block is found, the RBL loads the SBL from that block, ensuring a reliable boot process.

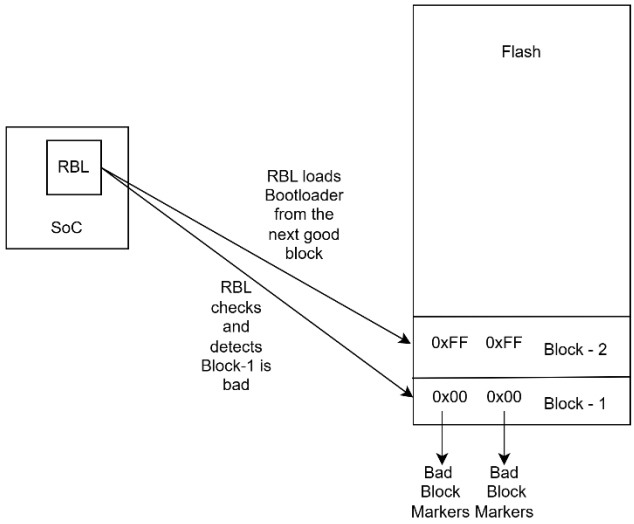


**Figure 3-1. Bad Block Skipping in RBL**

# 4 Bad Block Management in TI™ MCU+ SDK

The TI MCU+ SDK implements the Bad Block Management approach with the following key features:

1. Bad Block List Initialization
   - During flash driver initialization, the system scans all blocks in the device.
   - The spare area of each block's first page is read to check the bad block marker.
   - A bad block list table is implemented, tracking the status of every block.
2. Bad Block Skipping
   - All read, write, and erase operations automatically consult the bad block list.
   - When a bad block is encountered, the operation skips to the next good block.
3. Runtime Bad Block Marking
   - Program and erase operations are monitored for failure status.
   - When an operation fails, the affected block is immediately marked as bad. The following bits are set in the Status Register:
     – Program Failure (P-FAIL): The program failure bit is used to indicate whether the internally controlled program operation was executed successfully or timed out, by setting the P-FAIL bit as 0 or 1 respectively.
     – Erase Failure (E-FAIL): The erase failure bit is used to indicate whether the internally controlled erase operation was executed successfully or timed out, by setting the E-FAIL bit as 0 and 1 respectively.
   - The bad block marker of size 2 bytes is located at the beginning of the spare area, which starts at the pageSize offset within each page.
   - The in-memory bad block list is updated to reflect the new bad block.
   - Subsequent operations automatically avoid the newly marked block.

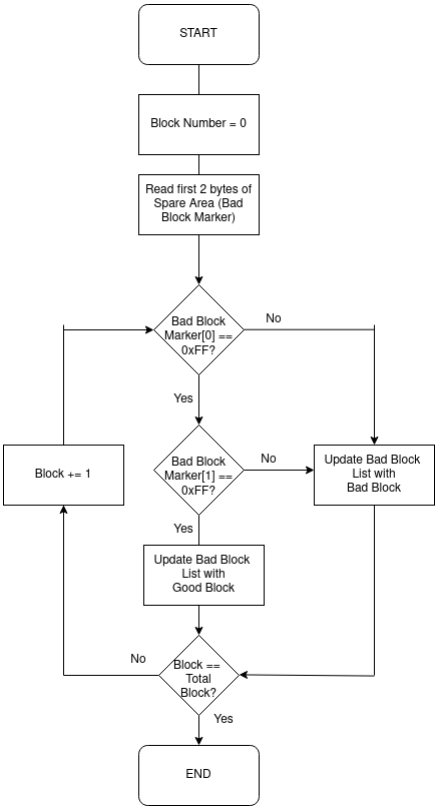


**Figure 4-1. Bad Block List Initialization**

## 4.1 Flash Read Operation Flow

1. Address Calculation:
   - The OSPI driver calculates the starting block number, page number, and offset within the page based on the requested byte offset. The number of pages to read is calculated based on the transfer length and page alignment.
2. Bad Block Skipping:
   - If bad block management is enabled, the driver checks the bad block list for the target block:
     - If the block is marked as good, the read operation is performed.
     - If the block is marked as bad, the block number and the page number is incremented to skip to the next block.
     - This check is repeated until a good block is found.
     - If the block number exceeds the total block count, a read error is returned.
3. Page-by-Page Reading.
4. Block Boundary Handling:
   - When the page counter crosses a block boundary, the driver recalculates the block number and checks the bad block list again. If the new block is marked as bad, the OSPI driver automatically skips to the next good block and adjusts the page number accordingly.
5. Completion:
   - The read operation continues until all requested data has been transferred. The driver returns success, and the application receives contiguous data even though the underlying physical blocks may be non-contiguous due to bad block skipping.

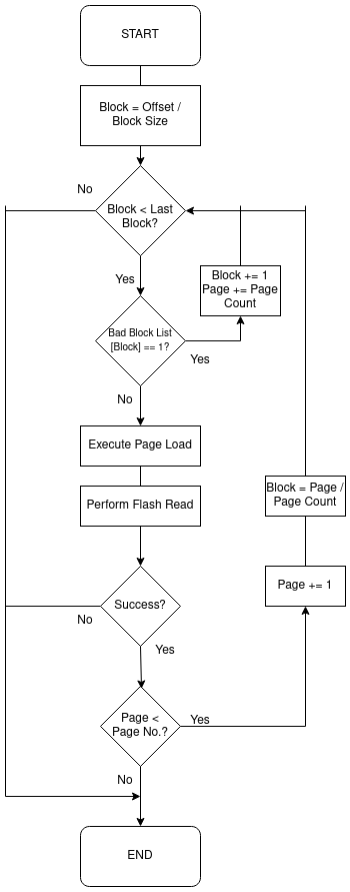


**Figure 4-2. Flash Read Flow**

## 4.2 Flash Write Operation Flow

1. Parameter Validation:
   - The write operation validates that the offset is page-aligned, which is required for Serial NAND Flash, and that the offset plus length does not exceed the flash device capacity.
2. Address Calculation:
   - The driver calculates the starting page address and block number from the byte offset.
3. Good Block Location:
   - If bad block management is enabled, the OSPI driver uses the bad block list to find the next good block starting from the calculated block number. If the target block is marked as bad, the system automatically advances to the next good block.
4. Page-by-Page Writing.
5. Program Failure Handling:
   - If the program status check indicates failure:
     - The block is then marked as bad by writing the bad block marker to the spare area.
     - The in-memory bad block list is updated to mark this block as bad.
     - If the write operation fails, a write error is returned.
6. Address Advancement:
   - After each successful page write, the driver increments the page address. When crossing a block boundary, the driver checks the bad block list and skips to the next good block if necessary.
7. Completion:
   - The write operation continues until all data has been written. The driver returns success when all pages have been programmed successfully.

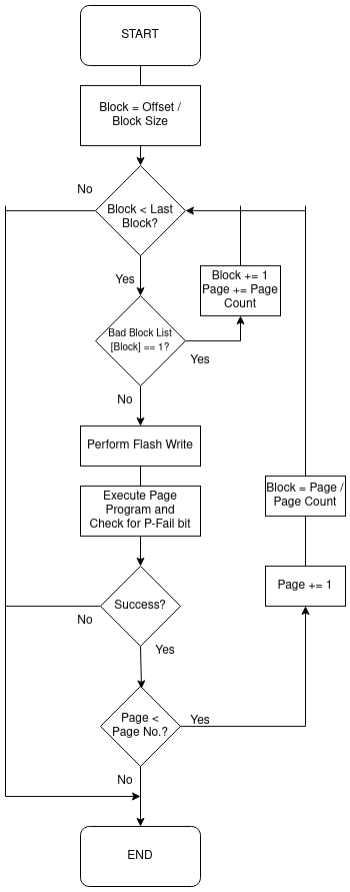


**Figure 4-3. Flash Write Flow**

## 4.3 Flash Erase Operation Flow

1. Bad Block Check:
   - If bad block management is enabled, the driver first checks if the target block is already marked as bad. If so, it immediately skips to the next block and checks again. This continues until a good block is found.
2. Execute Erase operation.
3. Erase Status Verification:
   - The driver polls the erase status register with a timeout to verify the erase operation completed successfully.
4. Erase Failure Handling:
   - If the erase status indicates failure and bad block management is enabled:
     - The block is marked as bad by writing the bad block marker to the spare area.
     - The in-memory bad block list is updated.
     - The block number is incremented to move to the next block.
     - The erase sequence is repeated on the next block.
5. Erase Failure Without BBM:
   - If bad block management is disabled and an erase fails, the operation returns an error immediately without retry.
6. Completion:
   - When a block is successfully erased, the operation exits the loop and returns success to the application.

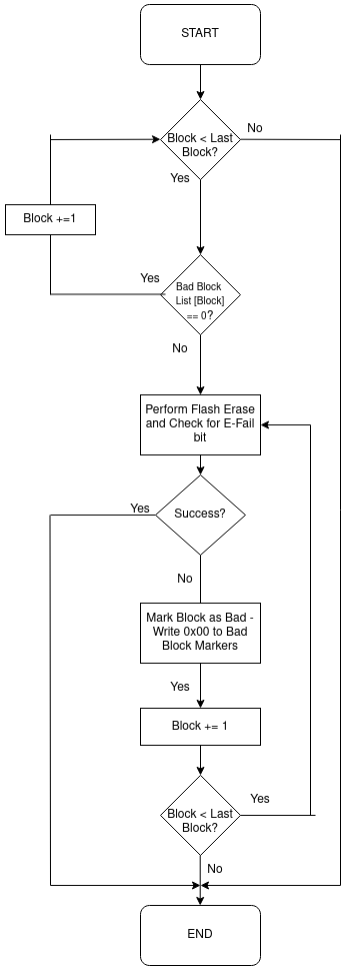


**Figure 4-4. Flash Erase Flow**

# 5 Bad Block Management in TI™ Processors SDK

The Linux Kernel implements Bad Block Management through the Memory Technology Device (MTD) subsystem and the Unsorted Block Images (UBI) with the following key features:

1. Bad Block Table Initialization
   - During flash driver initialization, the system creates an in-memory Bad Block table using a bitmap structure:
     - Each block's first page Out-Of-Band area is read.
     - The bad block marker consists of 2 bytes at typically the beginning of the OOB area.
     - A marker value of 0xFF for both bytes indicates a good block.
     - Any non-0xFF value (typically 0x00) indicates a factory-marked bad block or runtime bad block.
   - A Bad Block Table is constructed in memory using 2 bits per block to encode the status:
     - 00b = Good block, which indicates that the block is available for use.
     - 01b = Worn or runtime bad block which indicates that the erase or write operation failed during runtime.
     - 10b = Reserved block.
     - 11b = Factory bad block which is marked by the manufacturer.
   - On every system boot, the Bad Block Table is rebuilt by scanning the OOB markers of all the blocks present in the Flash.
2. Bad Block Skipping
   - All read, write, and erase operations automatically consult the in-memory Bad Block Table before accessing physical blocks.
   - The UBI layer maintains the Physical Erase Block (PEB) organization using three Red-Black trees:
     - Free Tree: Contains erased, good PEBs available for allocation, sorted by erase count for wear-leveling.
     - Used Tree: Contains PEBs currently storing volume data.
     - Erroneous Tree: Contains PEBs marked as bad or unreliable.
   - When the UBI layer needs to allocate a block for writing:
     - A PEB is selected from the free tree. A PEB is never selected from the erroneous tree.
     - The bad block status is verified by querying the BBT through the MTD layer.
     - If the block is marked bad, it is removed from the free tree, added to the erroneous tree, and a different PEB is selected.
   - The BBT lookup uses efficient bit manipulation for fast access:
     - Byte offset in BBT array = PEB number ÷ 4 (since each byte holds four 2-bit entries)
     - Bit position within byte = (PEB number mod 4) × 2
   - The 2-bit status code is the extracted using the following implementation: (BBT_byte >> bit_position) & 0x03
   - When a bad block is encountered during logical-to-physical address translation, the operation automatically skips to the next good block, making bad block avoidance transparent to upper layers.
3. Runtime Bad Block Marking
   - Program, erase, and read operations are continuously monitored for failure conditions through multiple mechanisms:
     - NAND controller status register checking after program and erase operations.
     - Error Correction Code (ECC) engine monitoring during read operations.
     - Wear-leveling algorithm tracking of marginal block behavior.
   - When an operation fails, the affected block is immediately marked as bad through the following sequence:
     - Program Failure: The P-FAIL bit of the NAND controller status register is checked after each program operation. If P-FAIL = 1 indicates that the program operation timed out or failed, triggering bad block marking.
     - Erase Failure: E-FAIL bit of the NAND controller status register is checked after each erase operation. If E-FAIL = 1 indicates that the erase operation failed, triggering bad block marking.
     - ECC Uncorrectable Error: When the number of bit errors in a page exceeds the ECC correction capability, that is, more than 8 bits for BCH-8 or 16 bits for BCH-16, the block is marked as failing.
   - The bad block marking procedure executes the following steps:

  - Update In-Memory BBT: The 2-bit status code for the failed block is changed from 00b, indicating a good block to 01b indicating runtime bad block.
  - Write Bad Block Marker to OOB: The bad block marker bytes (0x00,0x00) is written to the spare area or OOB of the first page of the failed block.
  - This ensures persistence across power cycles, as the in-memory BBT will be rebuilt from these markers on the next boot.
- The UBI layer updates its data structures:
  - The bad PEB is removed from the free tree or used tree, depending on where it was located.
  - The bad PEB is added to the erroneous tree to prevent future allocation.
  - If the bad PEB was storing volume data, the LEB-to-PEB mapping is updated to invalidate affected logical erase blocks.
- Subsequent operations automatically avoid the newly marked block, as it now resides in the erroneous tree and has status 01b in the BBT.

## 5.1 Flash Read Operation Flow

1. Address Translation:
   a. UBIFS receives a read request from the application and translates the file offset to a Logical Erase Block (LEB) number.
   b. UBI consults its LEB-to-PEB mapping table to determine which Physical Erase Block holds the requested data.
   c. UBI calculates the physical flash address by multiplying the PEB number by the erase block size and adding the page offset.
2. Bad Block Check:
   a. Before accessing physical flash, UBI verifies whether the target PEB is not marked bad.
   b. The MTD layer accesses the in-memory Bad Block Table, calculating byte offset and bit position.
   c. The 2-bit status code is extracted.
3. Flash Read Execution:
   a. If the block is good, MTD invokes the NAND controller driver's read function with the physical address.
   b. The controller driver programs hardware registers with the flash address and issues the read command.
   c. The NAND flash chip reads the page into its internal buffer and transfers data to system memory.
4. ECC Processing:
   a. The ECC engine processes the received data and ECC parity bytes from the OOB area.
   b. The engine performs syndrome calculation to detect bit errors.
   c. If errors are within correction capability, the engine corrects them, and the read succeeds.
   d. If errors exceed correction capability, the read fails, and the block is marked bad.
5. Completion:
   a. Valid data passes from MTD -> UBI -> UBIFS -> application.
   b. UBI validates sequence numbers and CRC values in headers.
   c. If uncorrectable errors occur, error handling attempts recovery from alternate copies or returns I/O error to application.

## 5.2 Flash Write Operation Flow

1. PEB Allocation:
   a. UBIFS determines whether a write operation requires allocating a new Logical Erase Block.
   b. UBI allocates a Physical Erase Block from the free tree, selecting a block with lower erase count for wear-leveling.
2. Pre-Write Bad Block Check:
   a. UBI verifies whether the selected PEB is not marked bad.
   b. MTD performs a BBT lookup to extract the 2-bit status code.
   c. If bad, the UBI removes the PEB from free tree, adds to erroneous tree, and selects a different PEB.
3. Write Preparation:
   a. UBI prepares data with UBI headers: volume ID, LEB number, sequence numbers, data CRC, header CRC.
   b. MTD calls the controller driver's program function with physical address and data buffer.
4. Flash Program Operation:
   a. Driver loads data into NAND chip's page buffer through the data port.
   b. Driver issues program command by writing to controller's command register.
5. Program Status Verification:
   a. Driver reads NAND status register checking P-FAIL bit.
   b. P-FAIL = 0 indicates that the program operation succeeded, and the driver returns success.
   c. P-FAIL = 1 indicates that the program operation failed and the driver returns error.
6. Success Handling:
   a. If write succeeds, the UBI updates LEB-to-PEB mapping table.
   b. The UBI updates PEB's erase count and moves PEB from free tree to used tree.
7. Failure Handling:
   a. If write fails, UBI immediately marks the block bad:
      i. Update in-memory BBT by changing the status from 00b, indicating a good block, to 01b, indicating a runtime bad block.
      ii. Write bad block marker (0x00) to the OOB of the first page.
   b. UBI removes failed PEB from free tree and adds to erroneous tree.
   c. UBI selects new PEB from free tree and retries write operation.

## 5.3 Flash Erase Operation Flow

1. Erase Request Context:
   a. UBI wear-leveling algorithm triggers erase operations during:
      i. Garbage collection: Reclaiming blocks with obsolete data.
      ii. Wear-leveling: Moving data to equalize erase count distribution.
      iii. Block recycling: Preparing used blocks for reallocation.
   b. UBI selects a block for erasing based on wear-leveling criteria.
2. Pre-Erase Bad Block Check:
   a. UBI queries the Bad Block Table with the target PEB number.
   b. MTD performs BBT lookup extracting the 2-bit status code.
   c. If the block is already marked bad, the UBI:
      i. skips the erase operation entirely.
      ii. removes the PEB from its current tree.
      iii. adds the PEB to the erroneous tree.
      iv. selects a different block if the operation requires continuing.
3. Flash Erase Execution:
   a. If the block is good, the UBI calls MTD erase function with block's physical address.
   b. MTD invokes controller driver's erase function.
   c. The driver then programs address registers with block address and issues the erase command to controller's command register.
4. Erase Completion and Status Check:
   a. The driver polls ready/busy status line or reads status register waiting for completion.
   b. After completion, driver reads status register for checking the E-FAIL bit.
   c. E-FAIL = 0 indicates that erase operation succeeded and the driver returns success.
   d. E-FAIL = 1 indicates that the erase failed and the driver returns error.
5. Erase Success Handling:
   a. If erase succeeds:
      i. UBI increments the PEB's erase count in tracking structures.
      ii. UBI moves the erased PEB to the free tree, making it available for allocation.
      iii. The block is now in clean erased state with all pages available for programming.
6. Erase Failure Handling:
   a. If erase fails:
      i. The UBI updates the in-memory BBT.
      ii. The bad block marker (0x00, 0x00) is written to the OOB area of the first page.
      iii. Update UBI data structures:
         1. The failed PEB is removed from the current tree.
         2. The failed PEB is added to the erroneous tree.
         3. The device capacity is updated, reflecting one fewer usable PEB.
   b. If erase was part of ongoing operation, UBI selects different block and continues.

## 6 Summary

By implementing Bad Block Management, embedded systems can leverage high-capacity NAND flash while maintaining data integrity throughout the device lifetime. The bad block management system abstracts the complexity of handling defective blocks, allowing applications to focus on their core functionality while the driver ensures reliable flash operation. This Application Note covers the implementation of Bad Block Management in the ROM Bootloader, TI™ MCU+ SDK, and the Processors Linux SDK.

## 7 References

1. Texas Instruments, MCU+ SDK and Processor SDK, software development kit.
2. Texas Instruments, AM62x Technical Reference Manual, technical reference manual.

# 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.



Last updated 10/2025