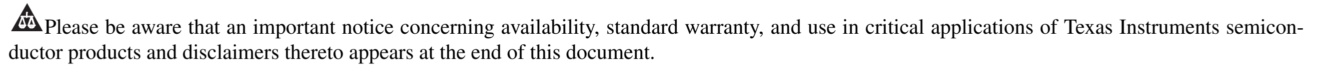
MSP MCU FRAM Utilities version 03.00.00.22

USER'S GUIDE

Copyright © Texas Instruments Incorporated.

Copyright

Copyright © Texas Instruments Incorporated. All rights reserved.



Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

Texas Instruments
Post Office Box 655303
Dallas, TX 75265
http://www.ti.com/msp430

Revision Information

This is version 03.00.00.22 of this document, last updated on January 17, 2017.

Table of Contents

Copyright 2
Revision Information 2
1 Introduction 5
2 Compute Through Power Loss (CTPL) 7
2.1 Introduction 7
2.2 Usage 8
2.2.1 Components 8
2.2.2 Debugging LPM3.5 and LPM4.5 Modes 10
2.2.3 Code Composer Studio (CCS) 11
2.2.4 IAR Embedded Workbench 13
2.3 API Reference 16
2.3.1 API Overview 16
2.3.2 Core API Set 16
2.3.3 Low Level 21
2.3.4 Peripherals 24
2.3.5 Benchmark 25
2.4 Examples 26
2.4.1 Examples Overview 26
2.4.2 LPM4.5 With GPIO Wakeup 27
2.4.3 LPM3.5 With RTC Wakeup 28
2.4.4 COMP_E Powerloss Monitor 29
2.4.5 ADC12_B Powerloss Monitor 30
2.5 Benchmarking 31
2.5.1 Overview 31
2.5.2 Configuration 33
3 LZ4 Compression 35
3.1 Introduction 35
3.2 Usage 36
3.2.1 Components 36
3.2.2 Performance Configuration 37
3.2.3 Command Line Utility 38
3.3 API Reference 39
3.3.1 API Overview 39
3.3.2 LZ4 39
3.3.3 xxHash 46
3.4 Examples 48
3.4.1 Examples Overview 48
3.4.2 Compress Text to LZ4 File 48
3.4.3 Compress Text with CRC16 Checksum 48
3.4.4 Decompress Data Stream 48
3.5 LZ4 Benchmarks 49
3.5.1 LZ4 Compression Ratio 49
3.5.2 LZ4 Compression Speed 51
4 Random Number Generator (RNG) 53
4.1 Introduction 53
4.2 API Reference 54
4.2.1 API Overview 54
4.2.2 RNG 54

4.3 Examples . . . 56
4.3.1 Examples Overview . . . 56
4.3.2 Generate Random Data . . . 56
4.3.3 Generate Random Data to CSV File . . . 56

5 Non-Volatile Storage (NVS) . . . 57
5.1 Introduction . . . 57
5.2 Features . . . 57
5.3 Storage Containers . . . 57
5.3.1 Data Storage . . . 57
5.3.2 Log Storage . . . 58
5.3.3 Ring Storage . . . 58
5.4 Memory Allocation . . . 58
5.4.1 FRAM . . . 58
5.4.2 Information Memory . . . 58
5.5 API Reference . . . 59
5.5.1 NVS . . . 59
5.5.2 NVS Data . . . 59
5.5.3 NVS Log . . . 63
5.5.4 NVS Ring . . . 66
5.5.5 NVS Support . . . 69
5.6 Examples . . . 71
5.6.1 Examples Overview . . . 71
5.6.2 Continuous Counter . . . 71
5.6.3 Application Configuration . . . 71
5.6.4 Data Logger . . . 71
5.6.5 Black Box Recorder . . . 71

IMPORTANT NOTICE . . . 72

1 Introduction

The Texas Instruments FRAM Utilities is a collection of embedded software utilities that leverage the ultra-low power and virtually unlimited write endurance of ferroelectric RAM (FRAM). The utilities are available for MSP430FRx FRAM microcontrollers and provide example code to help start application development.

Included are the following FRAM Utilities:

- Compute Through Power Loss (CTPL): A utility application programming interface (API) that enables ease of use with LPMx.5 low-power modes and a powerful shutdown mode that allows an application to save and restore critical system components when a power loss is detected.
- LZ4 Compression (LZ4): A lightweight compression utility based on the open source LZ4 compression standard and algorithm. The utility provides APIs for both compression and decompression on MSP FRAM microcontrollers and has been optimized for ultra-low power to enable data logging, over the air updates and more.
- Random Number Generator (RNG): Implementation of a counter mode deterministic random byte generator (CTR-DRBG) according to the NIST SP 800-90A Rev 1 specification. Random numbers are generated using seed information stored in the TLV tables and are unique to each device.
- Non-Volatile Storage (NVS): Library that make handling of non-volatile data easy and robust against intermittent power loss or asynchronous device resets. Includes three different storage containers for a wide range of applications.

2 Compute Through Power Loss (CTPL)

Introduction 7
Usage 7
API Reference 16
Examples 26
Benchmarking 31

2.1 Introduction

Compute Through Power Loss (CTPL) is a utility API set that leverages FRAM to enable ease of use with LPMx.5 low-power modes (LPM) and provides a powerful shutdown mode that allows an application to save and restore critical system components when a power loss is detected.

Traditional use of the LPM3.5 and LPM4.5 cause the application to reset when waking up and both application and peripheral state are not retained. The application must check for the LPMx.5 reset source at the start of the program and execute a separate branch of code if the device is waking up from a LPMx.5 mode. This often includes reinitializing both core system and application required peripherals in addition to initialization of global variables by the compiler defined c-start up function that is executed before the main program. This increases the start up time and increases the complexity of applications. As a result application programmers often avoid these low-power modes unless absolutely necessary.

The CTPL utility provides an easier solution for the application programmer. The included linker configuration files will place all application data sections into FRAM where they are retained through LPMx.5 low-power modes. The utility will also allocate FRAM storage used to save the state of the application and critical system peripherals. When entering into low-power modes with the CTPL utility the FRAM storage will be used to save the necessary components and the utility will put the device into the specific low power mode and wait for a device wakeup or reset. Upon device wakeup or reset the utility will intercept the reset and restore the application and peripheral state from the FRAM storage. After restoring the state the utility returns back to the application and the next line of code is executed, removing the need for the application programmer to check for a reset at the start of main.

Application execution using LPMx.5 modes and the CTPL utility can now be written using the same methods as LPM0-3 where the system state is retained. This enables existing applications to easily integrate the CTPL utility and begin using LPMx.5 modes in place of existing LPM0-3 modes and avoid rewriting complex application start up code.

Additionally the CTPL utility provides an API to safely save and restore context in the event of a powerloss. The utility will save the state of the application and critical system peripherals just like the LPMx.5 modes and then wait for the device to enter a BOR due to powerloss. A configurable parameter allows for a timeout for situations where the voltage ramps back up to operational levels. A device reset, power on or timeout will all restore the saved state and return to the application in the same manner as the LPMx.5 functions. See the CTPL examples section for powerloss monitor examples using an internal ADC12_B window comparator solution and an external COMP_E solution using a simple voltage divider to detect when power is lost.

2.2 Usage

Components . . . 8
Code Composer Studio (CCS) . . . 11
IAR Embedded Workbench . . . 13
Debugging LPM3.5 and LPM4.5 Modes . . . 10

2.2.1 Components

The CTPL utility consists of the following software components. Some of these are intended to be directly called from the application while others are internal to the utility implementation.

2.2.1.1 Core API Set

The CTPL Core API Set represents utility API's that the application can directly interface with. The simple API set includes the following functions:

- ctpl_init(): Initialize the CTPL library at the start of the system pre-init.
- ctpl_enterLpm35(): Save context, enter LPM3.5, restore context and return to the main application.
- ctpl_enterLpm45(): Save context, enter LPM4.5, restore context and return to the main application.
- ctpl_enterShutdown(): Save context, disable all interrupt sources, configure watchdog timeout and wait for BOR. Restore context and return to the main application on a device reset, power on or timeout.

See the Core API reference for complete API documentation.

2.2.1.2 Low Level

Low-level C and assembly functions that directly interface with the MSP430 to save the state and enter low power modes. These functions are called by the Core API Set and should not be invoked from the main application.

See the Low Level reference section for complete API documentation.

2.2.1.3 Peripheral

Peripheral specific functions to save and restore context. Each peripheral supported by the utility has a save, restore and epilogue function that can be defined by the CTPL device file based on peripheral availability and called by the Core API Set.

The CTPL utility currently supports the following peripherals. By default the core peripheral modules are enabled and the application peripheral modules are disabled.

- Core Peripherals

- System Resets, Interrupts, and Operating Modes, System Control Module (SYS)
- Power Management Module (PMM)
- Clock System (CS)
- 32-Bit Hardware Multiplier (MPY32)
- FRAM Controller (FRCTL)
- Memory Protection Unit (MPU)
- RAM Controller (RAMCTL)
- Digital I/O (PORT, PORT_INT)
- Watchdog Timer (WDT_A)
- Real-Time Clock (RTC)
- Real-Time Clock B (RTC_B)
- Real-Time Clock C (RTC_C)

■ Application Peripherals
- ADC Module (ADC)
- ADC Module (ADC10_B)
- ADC Module (ADC12_B)
- Capacitive Touch I/O (CAPTIO)
- Comparator D Module (COMP_D)
- Comparator E Module (COMP_E)
- CRC Module (CRC)
- CRC32 Module (CRC32)
- DMA Controller (DMAX_3)
- DMA Controller (DMAX_6)
- Enhanced Comparator (ECOMP)
- Enhanced Universal Serial Communication Interface (EUSCI_A)
- Enhanced Universal Serial Communication Interface (EUSCI_B)
- LCD Controller (LCD_C)
- LCD Controller (LCD_E)
- Smart Analog Combo (SAC)
- Timer (TIMER_A)
- Timer (TIMER_B)
- Trans-Impedance Amplifier (TRI)

See the Peripheral reference section for complete API documentation.

2.2.1.4 Device

Device specific C and linker configuration files. Every CTPL application needs to include the device C file that corresponds to the device being used. This device C file defines the peripherals that are saved and restored by the utility. Generally the LPMx.5 device wakeup time is significantly long enough that the peripheral restore routine has minimal impact on the overall wakeup time of the application, however certain peripherals can be excluded if they are not used in the application by editing this device C file. Additionally any CTPL application is required to use the device and IDE specific linker configuration file which places all read/write data into FRAM. Both of these files are included by default in the empty and example projects provided with the utility.

See the Code Composer Studio (CCS) or IAR Embedded Workbench section for IDE specific instruction on using the CTPL utility.

2.2.2 Debugging LPM3.5 and LPM4.5 Modes

Applications that use LPM3.5 and LPM4.5 modes can enable easier debugging by defining `CTPL_LPM_DEBUG` in the compiler predefined symbols. This will enable emulation of LPM3.5 and LPM4.5 modes and proper wakeup of the device upon receiving an interrupt. While emulating LPM3.5 and LPM4.5 modes the CPU is in active mode and polling for interrupts as a wakeup source. Upon receiving a wakeup interrupt the software performs a software reset and processes the event.

2.2.3 Code Composer Studio (CCS)

2.2.3.1 Creating an Empty CTPL Project

FRAM Utilities is a discoverable package in Code Composer Studio (CCS). Creating a new project with the complete CTPL library configured is as easy as selecting the "File -> New -> CCS Project" menu option and selecting the "Empty Project with FRAM Utilities" project template.

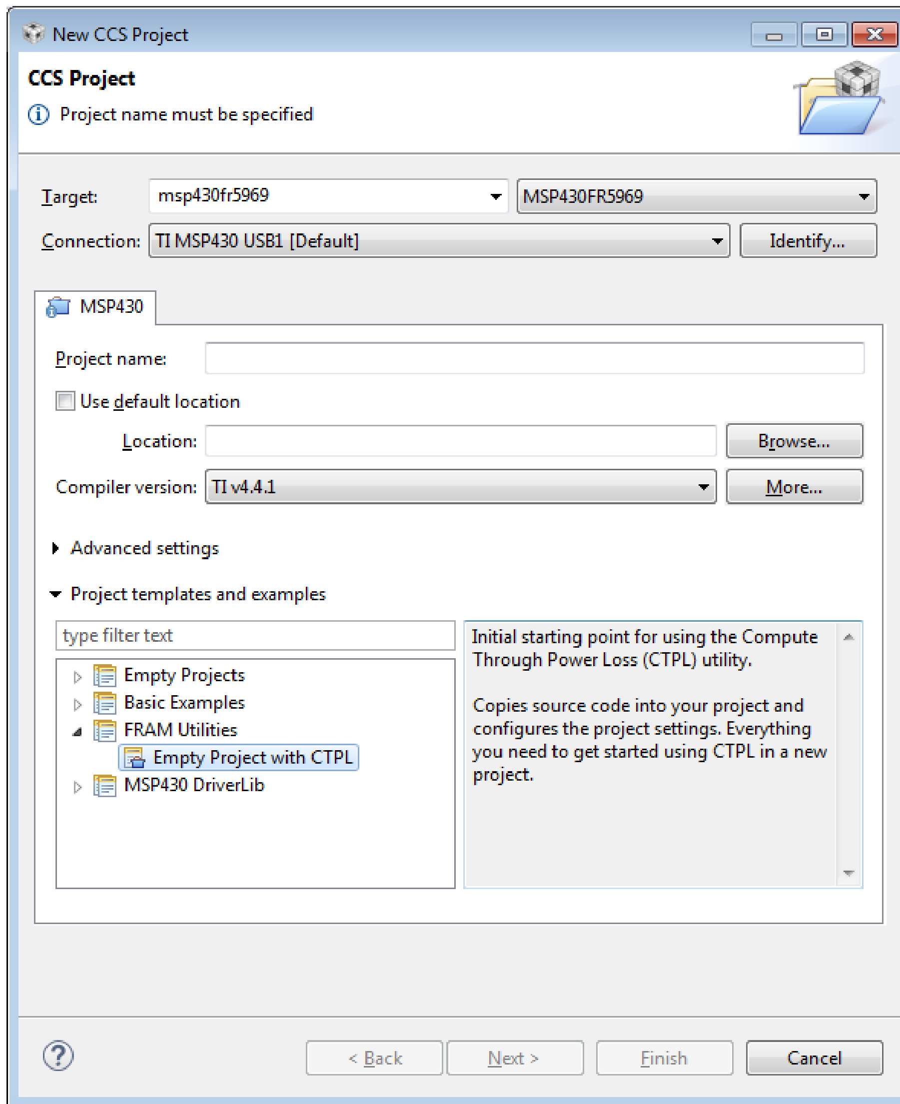


Figure 2.1: CCS new project wizard

2.2.3.2 Add CTPL to an Existing Application

The same project template can be used to apply the FRAM Utilities and CTPL settings and source code to an existing CCS project. Right click the project and select the "Source -> Apply Project

Template..." menu option and select the "Add Copy of FRAM Utilities to Project" project template.

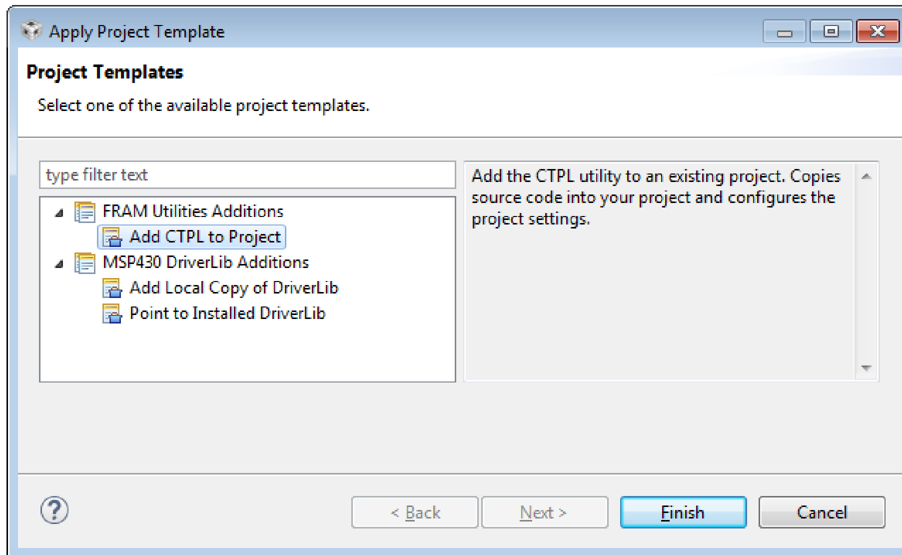


Figure 2.2: CCS apply project template

2.2.4 IAR Embedded Workbench

2.2.4.1 Opening the Examples

The CTPL utility provides an IAR Embedded Workbench workspace with preconfigured projects for each example. The workspace can be opened in IAR Embedded Workbench by double clicking the .eww file if windows associates this file type with IAR Embedded Workbench. Alternatively the workspace can be opened within IAR Embedded Workbench by navigating to and selecting the desired workspace in the "File -> Open -> Workspace" menu option.

2.2.4.2 Add CTPL to an Existing Application

Using the CTPL utility with IAR Embedded Workbench requires several different step to configure properly. The steps have been listed below and need to be followed closely to ensure proper integration with the existing application.

1. Add the CTPL source code to the project.
2. Add the CTPL include path to the project compiler options.

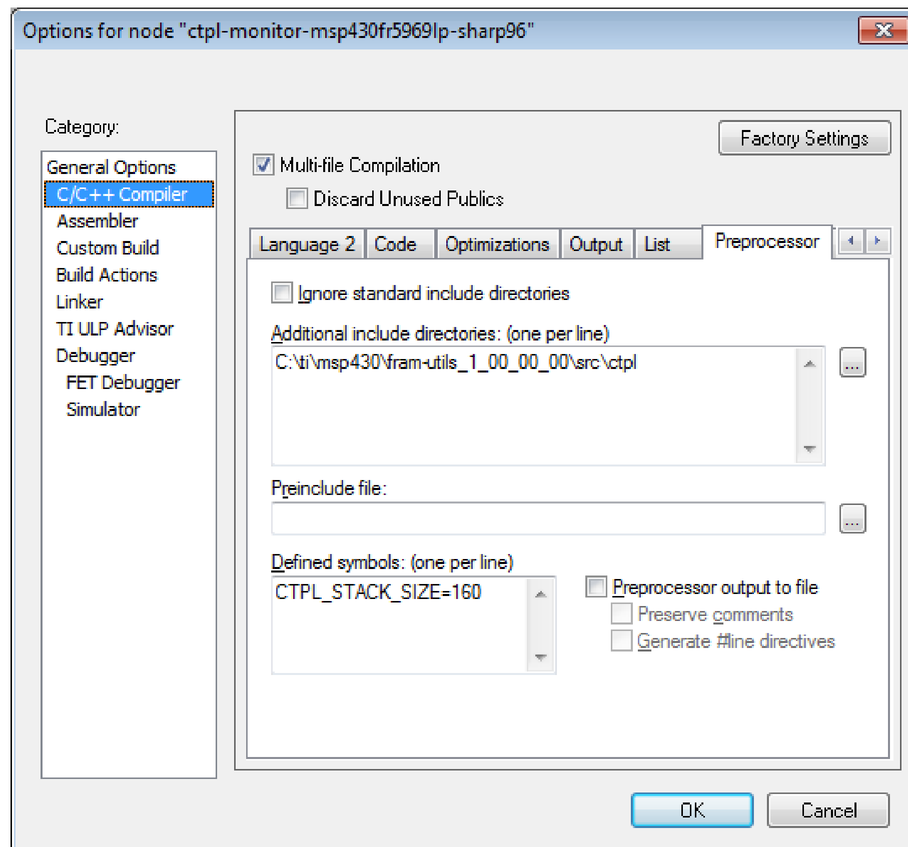


Figure 2.3: CTPL include path

3. Add the required predefined assembler symbols to the project assembler options.

(a) CTPL_STACK_SIZE is required and must be predefined to the same size as the configured stack size.

(b) __LARGE_CODE_MODEL__ is optional and should only be predefined if the project uses the large code model.

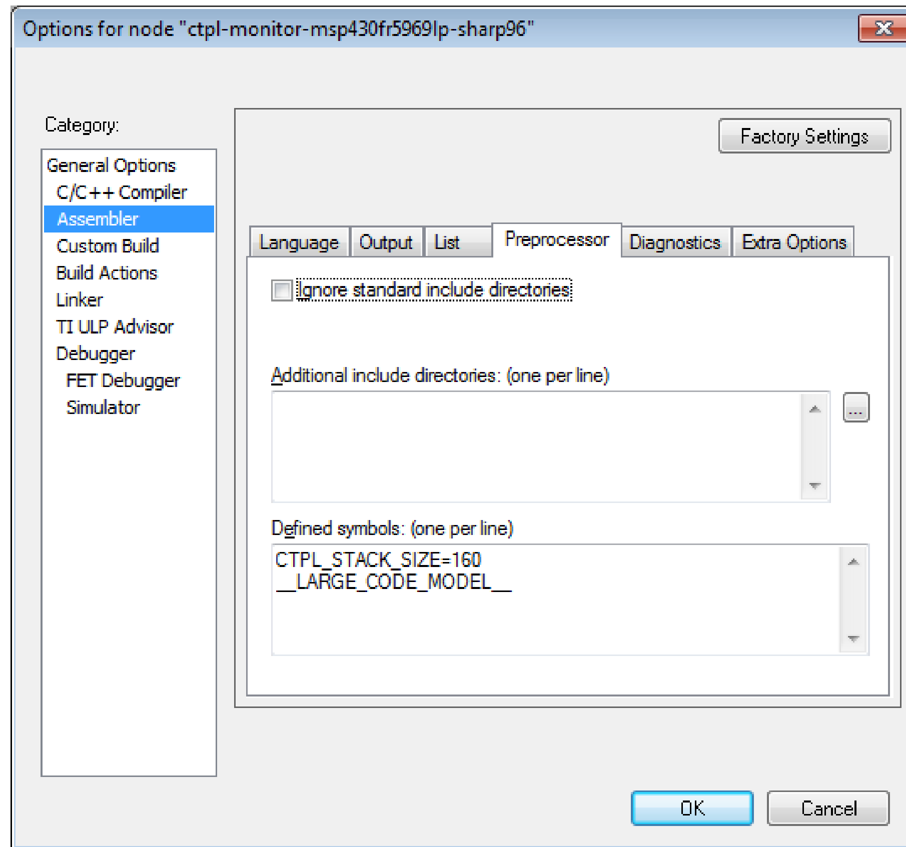


Figure 2.4: CTPL assembler options

4. Configure the project to use the device linker file (.xcl extension) in the project linker options.

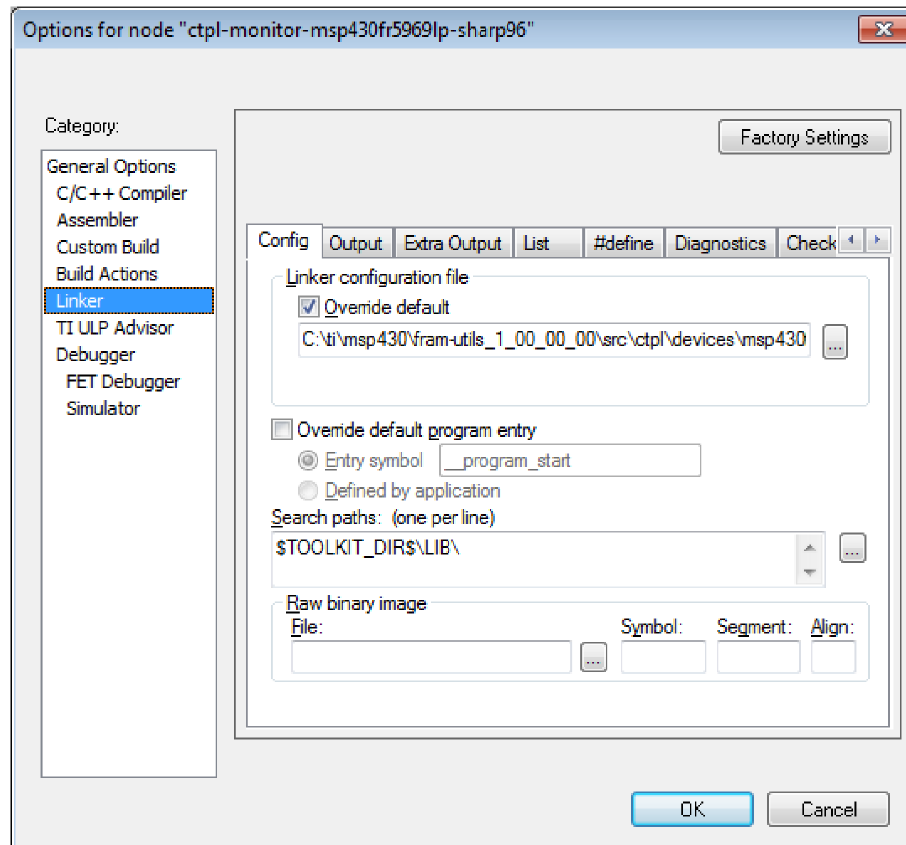


Figure 2.5: CTPL IAR linker file

2.3 API Reference

API Overview 16
Core API Set 16
Low Level 21
Peripherals 24
Benchmark 25

2.3.1 API Overview

The CTPL utility is designed to provide a simplified Core API set for use by the main application program. Methods outside of this API set have been documented below but are not intended to be modified or directly interfaced with by the main application program.

2.3.2 Core API Set

Macros

- #define CTPL_DISABLE_RESTORE_ON_RESET
- #define CTPL_ENABLE_RESTORE_ON_RESET
- #define CTPL_SHUTDOWN_TIMEOUT_1024_MS
- #define CTPL_SHUTDOWN_TIMEOUT_128_MS
- #define CTPL_SHUTDOWN_TIMEOUT_16_MS
- #define CTPL_SHUTDOWN_TIMEOUT_1_MS
- #define CTPL_SHUTDOWN_TIMEOUT_256_MS
- #define CTPL_SHUTDOWN_TIMEOUT_2_MS
- #define CTPL_SHUTDOWN_TIMEOUT_32_MS
- #define CTPL_SHUTDOWN_TIMEOUT_4_MS
- #define CTPL_SHUTDOWN_TIMEOUT_512_MS
- #define CTPL_SHUTDOWN_TIMEOUT_64_MS
- #define CTPL_SHUTDOWN_TIMEOUT_8_MS

Functions

- void ctpl_enterLpm35 (bool restoreOnReset)
- void ctpl_enterLpm45 (bool restoreOnReset)
- void ctpl_enterShutdown (uint16_t timeout)
- void ctpl_init (void)

2.3.2.1 Detailed Description

The following is a reference of all CTPL API's available for the application to use. The application should only directly interface with the function defined in `ctpl/ctpl.h` and listed below.

2.3.2.2 Macro Definition Documentation

2.3.2.2.1 #define CTPL_DISABLE_RESTORE_ON_RESET Do not allow the CTPL utility to restore a saved state if the device is reset or powered on from a cold start.

2.3.2.2.2 #define CTPL_ENABLE_RESTORE_ON_RESET Allow the CTPL utility to restore a saved state if the device is reset or powered on from a cold start.

Referenced by ctpl_enterShutdown().

2.3.2.2.3 #define CTPL_SHUTDOWN_TIMEOUT_1024_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 1024 milliseconds the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.2.4 #define CTPL_SHUTDOWN_TIMEOUT_128_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 128 milliseconds the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.2.5 #define CTPL_SHUTDOWN_TIMEOUT_16_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 16 milliseconds the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.2.6 #define CTPL_SHUTDOWN_TIMEOUT_1_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 1 millisecond the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.2.7 #define CTPL_SHUTDOWN_TIMEOUT_256_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 256 milliseconds the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.2.8 #define CTPL_SHUTDOWN_TIMEOUT_2_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 2 milliseconds the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.2.9 #define CTPL_SHUTDOWN_TIMEOUT_32_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 32 milliseconds the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.2.10 #define CTPL_SHUTDOWN_TIMEOUT_4_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 4 milliseconds the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.2.11 #define CTPL_SHUTDOWN_TIMEOUT_512_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 512 milliseconds the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.2.12 #define CTPL_SHUTDOWN_TIMEOUT_64_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 64 milliseconds the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.2.13 #define CTPL_SHUTDOWN_TIMEOUT_8_MS Timeout duration that can be passed to ctpl_enterShutdown(). If the device does not enter BOR after 8 milliseconds the watchdog timer will reset the device and cause a restore of the saved state.

2.3.2.3 Function Documentation

2.3.2.3.1 void ctpl_enterLpm35 (bool *restoreOnReset*) Save state and enter into low power mode LPM3.5.

LPM3.5 does not retain the settings of peripheral registers or RAM contents so these settings and states must be saved to non-volatile FRAM. This function will save the state of all the peripherals defined in the include device file, the context of the CPU and the active stack to non-volatile FRAM storage. After saving the state it is marked as valid so that it may be restored after wakeup and the function will enter into LPM3.5. When the device wakes up due to an interrupt or reset/power on event the ctpl_init() function will check if the state is valid and if it should be restored. The restoreOnReset argument determines if state context is restored on a device reset or power on, passing true will always restore the saved state where as passing false will only restore state on a LPM3.5 wakeup from interrupt (returning to the start of main if the device was reset). The saved peripheral states, CPU states and stack are restored from the FRAM storage and the function returns back to the application from where it was called. This function bypasses the need to check at device start up for a LPM3.5 wakeup and the application only needs to reinitialize peripherals that are not saved by the utility.

This API is functionally the same as ctpl_enterLpm45(). The actual low-power mode used (LPM3.5 or LPM4.5) is determined by the state of the RTC peripheral, LPM3.5 is used if the RTC is enabled and LPM4.5 is used if the RTC is disabled. For more information on low power modes refer to the device datasheet and user's guide.

Parameters

restoreOnReset	Allow the CTPL utility to restore a saved state if the device is reset or powered on from a cold start. Valid values are: ■ **CTPL_DISABLE_RESTORE_ON_RESET** ■ **CTPL_ENABLE_RESTORE_ON_RESET**

Returns

none

2.3.2.3.2 void ctpl_enterLpm45 (bool *restoreOnReset*) Save state and enter into low power mode LPM4.5.

LPM4.5 does not retain the settings of peripheral registers or RAM contents so these settings and states must be saved to non-volatile FRAM. This function will save the state of all the peripherals defined in the include device file, the context of the CPU and the active stack to non-volatile FRAM storage. After saving the state it is marked as valid so that it may be restored after wakeup and the function will enter into LPM4.5. When the device wakes up due to an interrupt or reset/power on event the ctpl_init() function will check if the state is valid and if it should be restored. The restoreOnReset argument determines if state context is restored on a device reset or power on, passing true will always restore the saved state where as passing false will only restore state on a LPM4.5 wakeup from interrupt (returning to the start of main if the device was reset). The saved peripheral states, CPU states and stack are restored from the FRAM storage and the function returns back to the application from where it was called. This function bypasses the need to check at device start up for a LPM4.5 wakeup and the application only needs to reinitialize peripherals that are not saved by the utility.

This API is functionally the same as ctpl_enterLpm35(). The actual low-power mode used (LPM3.5 or LPM4.5) is determined by the state of the RTC peripheral, LPM3.5 is used if the RTC is enabled and LPM4.5 is used if the RTC is disabled. For more information on low power modes refer to the device datasheet and user's guide.

Parameters

restoreOnReset	Allow the CTPL utility to restore a saved state if the device is reset or powered on from a cold start. Valid values are: ■ **CTPL_DISABLE_RESTORE_ON_RESET** ■ **CTPL_ENABLE_RESTORE_ON_RESET**

Returns

none

2.3.2.3.3 void ctpl_enterShutdown (uint16_t *timeout*) Save the state when power is lost.

Device shutdown does not retain the settings of peripheral registers or RAM contents so these settings and states must be saved to non-volatile FRAM. This function will save the state of all the peripherals defined in the include device file, the context of the CPU and the active stack to non-volatile FRAM storage. After saving the state it is marked as valid so that it may be restored after a reset or powering the device back on. All interrupt and wakeup sources are disabled and the device waits in active mode for the SVS to put the device into BOR. MCLK is configured to 4MHz and the SMCLK and WDT_A dividers are set based on the timeout parameter. In this state the only source of a wakeup is a device reset, power up or a shutdown timeout. In all three wakeup scenarios the state is restored and the application resumes. The saved peripheral states, CPU states and stack are restored from the FRAM storage and the function returns back to the application from where it was called.

When configuring the shutdown timeout parameter the device supply voltage and ramp conditions should be considered to avoid scenarios where voltage ramps down too slowly. If the timeout duration is not long enough the timeout will trigger a restore before the device enters the BOR state. In this scenario the restored image is no longer valid and the next power on will cause a device reset to the beginning of the main application. To prevent this a timeout duration should be selected so that sufficient time is provided for the supply voltage to ramp down and the timeout only triggers in the scenario where voltage ramps back up to operational levels.

This API provides a method for application programmers to efficiently save the application state

and shutdown the CPU when a power loss is detected and restore the applications state when the device regains power. The utility includes two examples demonstrating methods for monitoring the device voltage and detecting a power loss.

This API only saves and restores RTC_B and RTC_C registers that are not retained in LPMx.5 modes. In device shutdown the context of these other registers must be reinitialized if using these peripherals. See the device users guide for the complete list of RTC registers and details on which are retained.

Parameters

timeout	Configurable timeout for a reset if device does not enter BOR. Valid values are: ■ **CTPL_SHUTDOWN_TIMEOUT_1_MS** ■ **CTPL_SHUTDOWN_TIMEOUT_2_MS** ■ **CTPL_SHUTDOWN_TIMEOUT_4_MS** ■ **CTPL_SHUTDOWN_TIMEOUT_8_MS** ■ **CTPL_SHUTDOWN_TIMEOUT_16_MS** ■ **CTPL_SHUTDOWN_TIMEOUT_32_MS** ■ **CTPL_SHUTDOWN_TIMEOUT_64_MS** ■ **CTPL_SHUTDOWN_TIMEOUT_128_MS** ■ **CTPL_SHUTDOWN_TIMEOUT_256_MS** ■ **CTPL_SHUTDOWN_TIMEOUT_512_MS** ■ **CTPL_SHUTDOWN_TIMEOUT_1024_MS**

Returns

none

2.3.2.3.4 void ctpl_init (void) Initialize the CTPL utility.

This function initializes the utility and must be called at the start of the _system_pre_init function for CCS or the __low_level_init function for IAR. By default these functions are defined in ctpl_pre_init.c but some applications might have their own version of the function. In this case the ctpl_pre_init.c file can be omitted and the function called at the start of the application's low level function.

Returns

none

2.3.3 Low Level

Macros

- #define CTPL_MODE_BITS
- #define CTPL_MODE_LPM35
- #define CTPL_MODE_LPM45
- #define CTPL_MODE_LPMX5_WAKEUP
- #define CTPL_MODE_NONE

- #define CTPL_MODE_RESTORE_RESET
- #define CTPL_MODE_SHUTDOWN

Functions

- uint16_t ctpl_saveCpuStackEnterLpm (uint16_t mode, uint16_t timeout)

2.3.3.1 Detailed Description

The following is a reference of the CTPL low level functions. These functions are invoked by the Core API Set and should not be called from outside the utility.

2.3.3.2 Function Documentation

2.3.3.2.1 uint16_t ctpl_saveCpuStackEnterLpm (uint16_t *mode,* uint16_t *timeout*) Low level assembly function used to save the state and enter LPM.

Size of RAM contents to save to FRAM. By default this is set to the entire RAM contents for FR2xx and FR4xx devices and disabled for all other devices. This setting can be overriden by manually defining in the compiler options (–define=CTPL_RAM_SIZE=864).This assembly function saves the CPU state and stack into non-volatile FRAM before setting the state as valid and entering into the low-power mode defined by ctpl_mode. On device reset with a valid state ctpl_init will jump back to this function which restores the CPU state and stack from the FRAM copy. After restoring the state the function returns to the higher-level CTPL function that was invoked by the main application.

This function is only intended to be called from within the library code, the user does not need to invoke this function manually.

Parameters

mode	CTPL modes and flags. Valid flags are: ■ **CTPL_MODE_NONE** ■ **CTPL_MODE_LPM35** ■ **CTPL_MODE_LPM45** ■ **CTPL_MODE_SHUTDOWN** ■ **CTPL_MODE_RESTORE_RESET** ■ **CTPL_MODE_LPMX5_WAKEUP**
timeout	Configurable timeout for a reset if device does not enter BOR. Valid values are: ■ **CTPL_POWERLOSS_TIMEOUT_1_MS** ■ **CTPL_POWERLOSS_TIMEOUT_2_MS** ■ **CTPL_POWERLOSS_TIMEOUT_4_MS** ■ **CTPL_POWERLOSS_TIMEOUT_8_MS** ■ **CTPL_POWERLOSS_TIMEOUT_16_MS** ■ **CTPL_POWERLOSS_TIMEOUT_32_MS** ■ **CTPL_POWERLOSS_TIMEOUT_64_MS** ■ **CTPL_POWERLOSS_TIMEOUT_128_MS** ■ **CTPL_POWERLOSS_TIMEOUT_256_MS** ■ **CTPL_POWERLOSS_TIMEOUT_512_MS** ■ **CTPL_POWERLOSS_TIMEOUT_1024_MS**

Returns

mode CTPL mode and flags.

2.3.4 Peripherals

Data Structures

- struct ctpl_peripheral

Typedefs

- typedef struct ctpl_peripheral ctpl_peripheral
- typedef void(∗ ctpl_tFunction)(uint16_t baseAddress, uint16_t ∗storage, uint16_t mode)

Variables

- const ctpl_peripheral ∗const ctpl_peripherals [ ]
- const uint16_t ctpl_peripheralsLen

2.3.4.1 Detailed Description

The following is a reference of the CTPL peripheral functions. These functions are invoked by the Core API Set and should not be called from outside the utility.

2.3.4.2 Data Structure Documentation

2.3.4.2.1 struct ctpl_peripheral Structure defining how to save and restore a peripherals context. These structures are provided for each device in the included device-specific ctpl_∗.c file required when using the utility.

Data Fields

uint16_t	baseAddress	Peripheral base address.
ctpl_tFunction	epilogue	Optional function to run after clearing the LOCKLPM5 bit. If this function pointer is null the function will not be called.
ctpl_tFunction	restore	Function to restore peripheral context.
ctpl_tFunction	save	Function to save peripheral context.
uint16_t ∗	storage	Peripheral non-volatile storage.

2.3.4.3 Typedef Documentation

2.3.4.3.1 typedef struct ctpl_peripheral ctpl_peripheral Structure defining how to save and restore a peripherals context. These structures are provided for each device in the included device-specific ctpl_∗.c file required when using the utility.

2.3.4.3.2 typedef void(∗ ctpl_tFunction)(uint16_t baseAddress, uint16_t ∗storage, uint16_t mode) Function prototype for peripheral save, restore and epilogue functions.

Parameters

baseAddress	Peripheral base address.
storage	Peripheral non-volatile register storage.
mode	CTPL mode used.

Returns

none

2.3.4.4 Variable Documentation

2.3.4.4.1 const ctpl_peripheral∗ const ctpl_peripherals[ ] The device specific array of peripherals to save and restore. This symbol is defined in the device-specific ctpl_∗.c file included with the library.

2.3.4.4.2 const uint16_t ctpl_peripheralsLen Abstracted symbol for the length of the ctpl_peripherals array. This symbol is defined in the device-specific ctpl_∗.c file required when using the library.

2.3.5 Benchmark

The following is a reference of the CTPL benchmark function. These defines are used by the Core API Set and should not be referenced from outside the utility.

2.4 Examples

Examples Overview 26
LPM4.5 With GPIO Wakeup 27
LPM3.5 With RTC Wakeup 28
COMP_E Powerloss Monitor 29
ADC12_B Powerloss Monitor 30

2.4.1 Examples Overview

These examples demonstrate how to use the CTPL utility in several application use cases. The examples are implemented for all FRAM LaunchPad Development Kits and Experimenter Boards. See table below for supported hardware and examples.

Hardware	Examples			
	LPM4.5 GPIO	**LPM3.5 RTC**	**COMP_E Powerloss**	**ADC12_B Powerloss**
msp-exp430fr2311	✓	✓	X	X
msp-exp430fr4133	✓	✓	X	X
msp-exp430fr5739	✓	✓	X	X
msp-exp430fr5969	✓	✓	✓	✓
msp-exp430fr5994	✓	✓	✓	✓
msp-exp430fr6989	✓	✓	✓	✓

Table 2.1: Hardware support for CTPL examples

Using CCS and Resource Explorer it's easy to import and run the examples. Navigate to the CCS "View" menu and select "Resource Explorer (Examples)". Under the MSPWare package libraries select the FRAM-Utilities node and then CTPL node to view examples, user guides and release notes.

2.4.2 LPM4.5 With GPIO Wakeup

This example is an adaptation of the C code example msp430fr59xx_lpm4-5_01 and demonstrates how to enter LPM4.5 and wakeup from a GPIO interrupt. The example will turn on P4.6 and enter into LPM4.5. When P1.1 (S2 on MSP-EXP430FR5969) transitions from high to low the example will turn off P4.6 to indicate the device is no longer in LPM4.5 and blink P1.0 forever.

By using the compute through power loss (CTPL) library the original example code is greatly simplified. The peripherals are initialized once at the start of the application and the library will save the peripheral and application state in FRAM before entering LPM. Upon wakeup from LPM the peripheral and application state is restored and the code continues execution from the next line of code.

```
//  ACLK = VLOCLK, MCLK = SMCLK = DCO = ~1MHz
//
//           MSP-EXP430FR5969
//           ---------------
//       /|\|               |
//        | |           P1.0|---> LED2
//        --|RST        P4.6|---> LED1
//          |               |
//          |           P1.1|<--- S2 push-button
//          |               |
```

2.4.3 LPM3.5 With RTC Wakeup

This example is an adaptation of the C code example msp430fr59xx_lpm3-5_02 and demonstrates how to use RTC_B as an interval wakeup in LPM3.5. The example will toggle P4.6 after initialization to indicate a device start up and then enter LPM3.5 with interrupts enabled. The RTC interrupt will wake the device up every two seconds and toggle P1.0.

By using the compute through power loss (CTPL) library the original example code is greatly simplified. The peripherals are initialized once at the start of the application and the library will save the peripheral and application state in FRAM before entering LPM. Upon wakeup from LPM the peripheral and application state is restored and the code continues execution from the next line of code.

```
//  ACLK = 32.768kHz, MCLK = SMCLK = DCO = ~1MHz
//
//           MSP-EXP430FR5969
//           ---------------
//       /|\|            XIN|-
//        | |               | 32kHz
//        --|RST        XOUT|-
//          |               |
//          |           P1.0|--> LED2
//          |           P4.6|--> LED1
//          |               |
```

2.4.4 COMP_E Powerloss Monitor

This example demonstrates how to use the COMP_E peripheral and an external voltage divider to actively monitor supply voltage and detect when power is lost. The comparator is configured with a 1.5V reference and an external voltage divider provides Vcc/2 to the input pin (P1.5/C5). When Vcc/2 drops below the 1.5V reference (meaning Vcc is below 3.0V) the comparator interrupt service routine will disable the comparator monitor and invoke the ctpl_enterShutdown API. This API will save the application and peripheral state and waits for the device to enter BOR with a 64ms timeout. The device will restore application and peripheral state when power is restored and continue execution from the next line of code.

The main application will blink LED2 with incremental counts, resetting after four blinks. The power supply can be removed (by disconnecting the USB cable or unplugging the jumpers connecting the on-board emulator to the device) after a specific count of blink and then reapplied to verify that context was saved.

```
//  ACLK = VLOCLK, MCLK = SMCLK = DCO = ~1MHz
//
//           MSP-EXP430FR5969
//           ---------------
//       /|\|           P1.7|---> Vcc
//        | |       (C5)P1.5|---> Vcc/2 (350k/350k voltage divider)
//        --|RST        P1.4|---> GND
//          |               |
//          |           P1.0|---> LED2
//          |               |
```

2.4.5 ADC12_B Powerloss Monitor

This example demonstrates how to use the ADC12_B battery monitor and window comparator to actively monitor supply voltage and detect when power is lost. The ADC12_B peripheral is configured with a 2.0V reference voltage and the internal battery monitor channel provides Vcc/2. The ADC12_B low side window comparator is configured to trigger the interrupt when Vcc reaches ADC_MONITOR_THRESHOLD, 3.0V by default. The high side window comparator is set to ADC_MONITOR_THRESHOLD + 0.1V to ensure the device has reached a stable voltage before enabling the monitor. When the high side interrupt is triggered it is disabled and the low side interrupt is enabled to begin actively monitoring Vcc. When power loss is detected the device will invoke the ctpl_enterShutdown API which saves the application and peripheral state and waits for the device to enter BOR with a 64ms timeout. The device will restore application and peripheral state when power is restored and continue execution from the next line of code.

The main application will blink LED2 with incremental counts, resetting after four blinks. The power supply can be removed (by disconnecting the USB cable or unplugging the jumpers connecting the on-board emulator to the device) after a specific count of blink and then reapplied to verify that context was saved.

```
//  ACLK = VLOCLK, MCLK = SMCLK = DCO = ~1MHz
//
//           MSP-EXP430FR5969
//           ---------------
//       /|\|               |
//        | |               |
//        --|RST        P1.0|---> LED2
//          |               |
//          |               |
```

2.5 Benchmarking

Overview 31
Configuration 33

2.5.1 Overview

The CTPL utility can be benchmarked by defining `CTPL_BENCHMARK` in the compiler and assembler predefined symbols. When this symbol is defined the code will toggle a single pin to indicate the CTPL function has started. Once the state has been saved the software will toggle the benchmark pin to indicate the end of the CTPL function. The ctpl_enterShutdown() function will continue to toggle the benchmark pin inside the software loop while waiting for the device to enter a BOR. The repeated pin toggle provides a measurement of how long the CPU can run before complete power is lost and the device shuts down to help select the right timeout parameter.

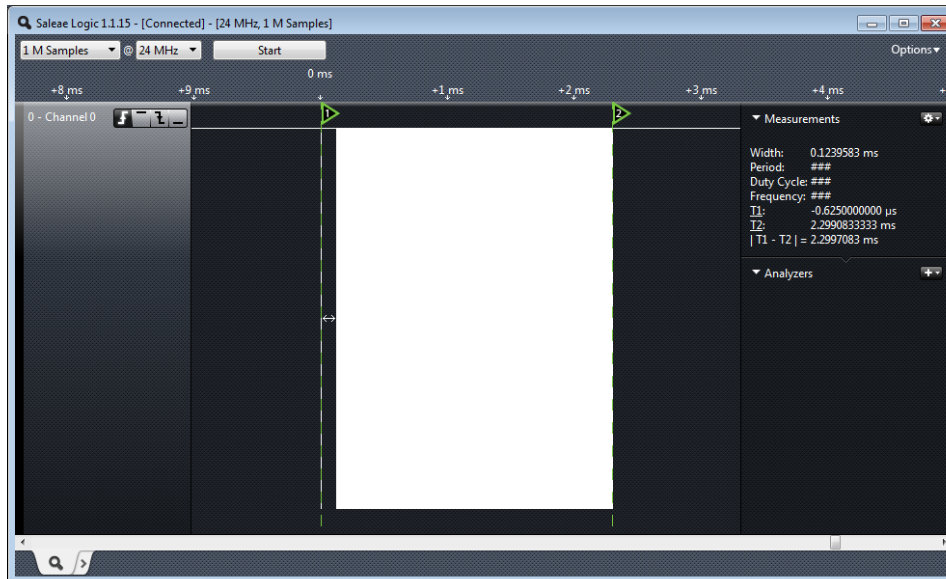


Figure 2.6: Benchmark of the ctpl_enterShutdown() function when power is lost (8MHz CPU clock)

The above screen capture shows the ctpl_enterShutdown() API when power is lost on a MSP430FR5969 device running at 8MHz with all available peripherals saved (a total of eleven, see peripheral usage section for the complete peripheral list). In this example the "Width" measurement is the total time the API needs to save the state of the peripherals, stack and CPU. The "|T1 - T2|" measurement indicates the life of the CPU before complete power is lost. The second measurement will be dependant on both the hardware design and the software configuration of the device (active peripherals when entering API). In scenarios where power is lost it's best practice to shut down any active peripherals before calling the API to conserve the remaining energy.

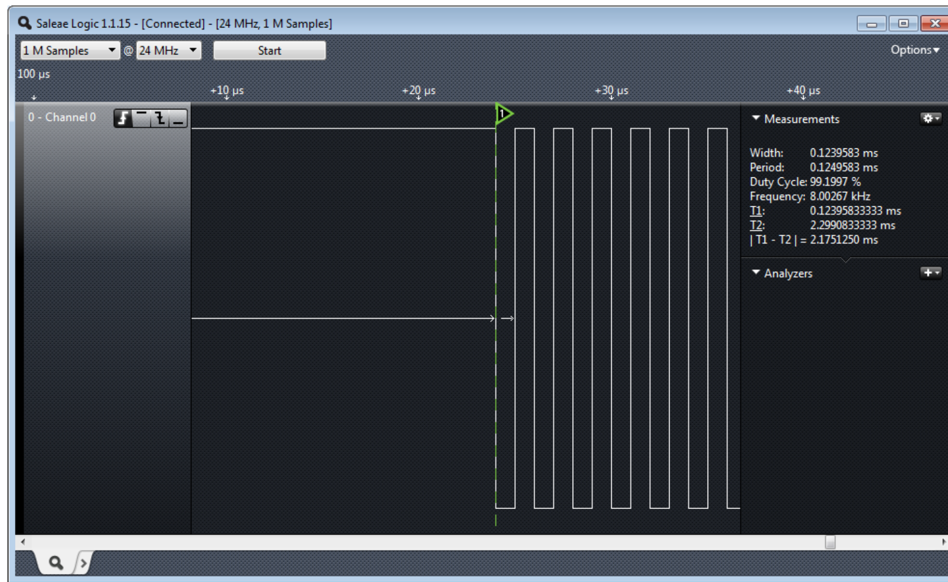


Figure 2.7: Close up view of benchmark pin toggle

2.5.2 Configuration

The pin used for the benchmark is defined in `ctpl_benchmark.h`. By default P4.6 is used, LED1 on the MSP430-EXP430FR5969 LaunchPad. This pin can be change to any available GPIO by editing this file and changing the pin and port registers used.

See the Benchmark API reference for more on configuration.

3 LZ4 Compression

Introduction . . . 35
Usage . . . 35
API Reference . . . 39
Examples . . . 48
Benchmarking . . . 49

3.1 Introduction

The LZ4 compression utility is a lightweight compression library optimized for ultra-low-power MSP microcontrollers. The software is based on the open source LZ4 specification and algorithm and is compatible with existing LZ4 software and files.

Traditional compression formats such as .gzip, .zip and .7z are optimized for high compression ratio with slow compression and fast decompression performance. These formats are typically used for file transfer where there is a single source and multiple destinations (such as file download). Compressing the files is a one time cost and it's beneficial to trade compression speed for a higher ratio. While compression may be very slow decompression is typically very fast, up to 100x faster, due to the nature of only needing to unroll the compressed file. While these algorithms are possible to run on an ultra-low-power microcontroller it would typically be impractical and take a significant amount of energy to enable both compression and decompression.

The LZ4 compression algorithm on the other hand is designed for both fast compression and decompression with less focus on compression ratio. This format is well suited for applications with point to point transfer of data such as in large data centers where reducing transfer time is desired but not at the cost of significantly increasing processing power to run compression. This makes LZ4 an excellent choice for an ultra-low-power embedded device where compression and decompression are equally important to target applications. The LZ4 algorithm provides the perfect balance of compression performance and functionality for a wide range of embedded applications such as data logging, expanded data storage, over-the-air data transfers and more.

The LZ4 full specification and open source software can be obtained at http://www.lz4.org.

3.2 Usage

Components . . . 36
Configuration . . . 37
Command Line Utility . . . 38

3.2.1 Components

The LZ4 compression utility consists of the following software components.

3.2.1.1 LZ4 API Set

The LZ4 API set included both compression and decompression API's that can be called directly from the main application.

- lz4_compress(): Compress a block of data to a LZ4 frame.
- lz4_compressBlock(): Compress a block of data to a single LZ4 block without framing.
- lz4_decompress(): Decompress a LZ4 frame to a block of data.
- lz4_decompressBlock(): Decompress a LZ4 block to a block of data.
- lz4_getContentSize(): Get the content size of a compressed LZ4 frame.

See the LZ4 API reference for complete API documentation.

3.2.1.2 xxHash API Set

The xxHash API set includes a single function for computing the xxHash of a data block. This function is provided to allow comparability with the official LZ4 framing structure however is not very efficient to run on MSP430 microcontrollers. If a checksum is necessary the application can use an available CRC module and the LZ4 block API's to create a custom framing scheme. See lz4_ex2_custom_frame for an example of how to create this.

- xxhash_compute(): Compute xxhash checksum for a block of data.

See the xxHash API reference for complete API documentation.

3.2.2 Performance Configuration

The following are options for configuring the performance of LZ4 compression. There are no performance configuration options for decompression.

3.2.2.1 Hash Table Size

The LZ4 compression API's use a hash table to find matches in the input data. A larger hash table provides a larger window of past history and a higher chance of finding a match (thus better compression ratio). The hash table size must be a power of two with a maximum size of 32768 bytes and is passed as the log2() of the table size. It is recommended to use a minimum size of 1024 bytes (hashLog2Size = 10) but some data sets may see a large improvement increasing the hash table size.

3.2.2.2 Hash Table Function

The LZ4 hash table function used for compression can be configured based on peripheral availability. By default the utility will use a multiplication based hash function, making use of the 32-bit hardware multiplier (MPY32) peripheral when available. For higher performance the compression functions can be configured to use CRC16 as a hashing function by defining `LZ4_CRC16_HASH`. When available the CRC16 or CRC32 peripheral will be used, see the LZ4 compression speed section for benchmarks of MPY32 versus CRC16 hash function performance.

3.2.3 Command Line Utility

The LZ4 open source project provides a command line utility that can be used to compress and decompress .lz4 files on another platform such as a personal computer. Some examples of how this can be used in combination with the LZ4 compression utility API's:

- Data logging: Store compressed data in FRAM, transmit to a host processor and decompress the data.
- Extended data storage: Compress program data on a personal computer with maximum compression and extract data at runtime on MSP MCU.
- Over the air update: Compress an entire MSP MCU program with maximum compression and transmit the data to a device for firmware upgrade.

The LZ4 source code and makefile to build the command line utility can be obtained with the latest release at http://www.lz4.org.

3.2.3.1 Windows

A pre-built command line utility is provided with this release. An example of invoking this command to compress a file with maximum compression settings is shown below.

```
${INSTALL_DIR}/tools/lz4.exe -9 data.txt
```

3.2.3.2 Linux

The command line utility can be installed on linux systems by running the following app-get command.

```
sudo apt-get install liblz4-tool
```

3.2.3.3 Mac

Mac users can download the latest release from http://www.lz4.org or extract the included release version and run Make.

3.3 API Reference

API Overview 39
LZ4 39
xxHash 46

3.3.1 API Overview

The LZ4 compression utility is designed to provide a simplified API set for use by the main application program.

3.3.2 LZ4

Data Structures

- struct lz4_compressBlockParams
- struct lz4_compressParams
- struct lz4_decompressBlockParams
- struct lz4_decompressParams
- struct lz4_stream_decompressBlockParams
- struct lz4_stream_decompressBlockState

Macros

- #define LZ4_COMPRESS_MAX_SIZE(n)

Typedefs

- typedef struct lz4_compressBlockParams lz4_compressBlockParams
- typedef struct lz4_compressParams lz4_compressParams
- typedef struct lz4_decompressBlockParams lz4_decompressBlockParams
- typedef struct lz4_decompressParams lz4_decompressParams
- typedef struct lz4_stream_decompressBlockParams lz4_stream_decompressBlockParams
- typedef struct lz4_stream_decompressBlockState lz4_stream_decompressBlockState

Enumerations

- enum lz4_status {
  LZ4_SUCCESS, LZ4_PARTIAL_SUCCESS, LZ4_NO_CONTENT_SIZE, LZ4_FRAMING_ERROR,
  LZ4_BLOCK_CHECKSUM_ERROR, LZ4_CONTENT_CHECKSUM_ERROR }

- enum lz4_stream_state {
  LZ4_BLOCK_SIZE, LZ4_TOKEN, LZ4_LITERAL_LENGTH, LZ4_LITERAL,
  LZ4_MATCH_LENGTH, LZ4_MATCH_OFFSET_LOW, LZ4_MATCH_OFFSET_HIGH }

Functions

- uint32_t lz4_compress (const lz4_compressParams ∗params, lz4_status ∗status)
- uint32_t lz4_compressBlock (const lz4_compressBlockParams ∗params, lz4_status ∗status)
- uint32_t lz4_decompress (const lz4_decompressParams ∗params, lz4_status ∗status)
- uint32_t lz4_decompressBlock (const lz4_decompressBlockParams ∗params, lz4_status ∗status)
- uint32_t lz4_getContentSize (const uint8_t ∗src, lz4_status ∗status)
- uint32_t lz4_stream_decompressBlock (lz4_stream_decompressBlockState ∗state, const void ∗data, uint16_t length, lz4_status ∗status)
- void lz4_stream_decompressBlockInit (const lz4_stream_decompressBlockParams ∗params, lz4_stream_decompressBlockState ∗state, lz4_status ∗status)

3.3.2.1 Detailed Description

The following is a reference of all LZ4 compression and decompression API's available for the application to use.

3.3.2.2 Data Structure Documentation

3.3.2.2.1 struct lz4_compressBlockParams Compression parameters for a single LZ4 block.

Data Fields

bool	addBlockChecksum	Add checksum to each compressed block. Decreases compression performance. Valid values are: ■ false ■ true
void ∗	dst	Pointer to the destination data buffer to store compressed data.
uint16_t	hashLog2Size	Power of two used to determine the hash table size.
void ∗	hashTable	Pointer to memory block with pow2(hashLog2Size) bytes allocated.
uint32_t	length	Length of source data buffer.
const void ∗	src	Pointer to the source data buffer to compress.

3.3.2.2.2 struct lz4_compressParams Compression parameters for a LZ4 frame.

Data Fields

bool	addBlockChecksum	Add checksum to each compressed block. Decreases compression performance. Valid values are: ■ false ■ true
bool	addContentChecksum	Add checksum of original source data buffer. Decreases compression performance. Valid values are: ■ false ■ true
bool	addContentSize	Add total content size to LZ4 frame. Increases total frame size by 8 bytes. Valid values are: ■ false ■ true
void *	dst	Pointer to the destination data buffer to store compressed data.
uint16_t	hashLog2Size	Power of two used to determine the hash table size.
void *	hashTable	Pointer to memory block with pow2(hashLog2Size) bytes allocated.
uint32_t	length	Length of source data buffer.
const void *	src	Pointer to the source data buffer to compress.

3.3.2.2.3 struct lz4_decompressBlockParams

Decompression parameters for a single LZ4 block.

Data Fields

void *	dst	Pointer to the destination data buffer to store compressed data.
uint32_t	dstLength	Length of the destination data buffer.
const void *	src	Pointer to the source data buffer to decompress.
bool	verifyBlockChecksum	Enable verification of block checksum if present. Valid values are: ■ false ■ true

3.3.2.2.4 struct lz4_decompressParams

Decompression parameters for a LZ4 frame.

Data Fields

void *	dst	Pointer to the destination data buffer to store compressed data.
uint32_t	dstLength	Length of the destination data buffer.
const void *	src	Pointer to the source data buffer to decompress.
bool	verifyBlockChecksum	Enable verification of block checksum if present. Valid values are: ■ false ■ true
bool	verifyContentChecksum	Enable verification of content checksum if present. Valid values are: ■ false ■ true

3.3.2.2.5 struct lz4_stream_decompressBlockParams

Decompression parameters for a streaming LZ4 block.

Data Fields

bool	containsBlockSize	Compressed LZ4 data contains block size. ■ false ■ true
void *	dst	Pointer to the destination data buffer to store compressed data.
int32_t	dstLength	Length of the destination data buffer.

3.3.2.2.6 struct lz4_stream_decompressBlockState

Decompression state for a streaming LZ4 block.

Data Fields

uint32_t	dstLength	Length of the destination data buffer.
uint8_t *	dstOrigin	Destination buffer pointer origin.
uint8_t *	dstPtr	Pointer to the destination data buffer to write.
uint16_t	literalLength	Length of literals.
uint16_t	matchLength	Length of match.
uint16_t	matchOffset	Offset address of match.
lz4_stream_state	state	Decompress stream state.

3.3.2.3 Enumeration Type Documentation

3.3.2.3.1 enum lz4_status

LZ4 status return types.

Enumerator

- ***LZ4_SUCCESS*** Successful operation.
- ***LZ4_PARTIAL_SUCCESS*** Data was partially decompressed due to insufficient space.
- ***LZ4_NO_CONTENT_SIZE*** Content size is not present in LZ4 frame header.
- ***LZ4_FRAMING_ERROR*** Error in frame header.
- ***LZ4_BLOCK_CHECKSUM_ERROR*** Incorrect block checksum.
- ***LZ4_CONTENT_CHECKSUM_ERROR*** Incorrect content checksum.

3.3.2.3.2 enum lz4_stream_state

LZ4 streaming API state.

Enumerator

- ***LZ4_BLOCK_SIZE*** Next byte is the token.
- ***LZ4_TOKEN*** Next byte is the token.
- ***LZ4_LITERAL_LENGTH*** Next byte is the literal length.
- ***LZ4_LITERAL*** Next byte is a literal character.
- ***LZ4_MATCH_LENGTH*** Next byte is the match length.
- ***LZ4_MATCH_OFFSET_LOW*** Next byte is the low match address offset.
- ***LZ4_MATCH_OFFSET_HIGH*** Next byte is the high match address offset.

3.3.2.4 Function Documentation

3.3.2.4.1 uint32_t lz4_compress (const lz4_compressParams * *params*, lz4_status * *status*)

Compress a block of data to a LZ4 frame.

Compress a block of data using LZ4 compression and add LZ4 framing. This API will compress data to a valid LZ4 file and contains several parameters to enable or disable features of the LZ4 framing specification such as content and block checksum using the xxHash algorithm or block size. See the lz4_compressParams structure documentation for more details about the available parameters.

A block compressed with this method can be saved as a binary .lz4 file and extracted using the LZ4 command line utility.

Parameters

params	Pointer to the LZ4 compression parameter structure.
status	Pointer to a LZ4 status that will contain the result status.

Returns

The total compressed frame size.

3.3.2.4.2 uint32_t lz4_compressBlock (const lz4_compressBlockParams * *params,* lz4_status * *status*) Compress a block of data to a single LZ4 block without framing.

Compress a block of data using only LZ4 compression and block format. This API can be used to compress data and create a custom framing scheme using an alternative checksum method. The LZ4 block format has a single parameter for enabling block checksum computation with the xxHash algorithm. The block checksum is computed on the compressed data block. See the lz4_compressBlockParams structure documentation for more details about the available parameters.

Parameters

params	Pointer to the LZ4 compression block parameter structure.
status	Pointer to a LZ4 status that will contain the result status.

Returns

The total compressed block size.

Referenced by lz4_compress().

3.3.2.4.3 uint32_t lz4_decompress (const lz4_decompressParams * *params,* lz4_status * *status*) Decompress a LZ4 frame to a block of data.

Decompress a LZ4 frame to an uncompressed data block. This API contains parameters to enable checking of content and block checksum. When enabled the xxHash algorithm is used to verify the checksum. While not required it is the application programmers responsibility to determine if validating the checksum is necessary. See the lz4_decompressParams structure documentation for more details about the available parameters.

A .lz4 file compressed with the LZ4 command line utility can be decompressed using this API.

Parameters

params	Pointer to the LZ4 decompression parameter structure.
status	Pointer to a LZ4 status that will contain the result status.

Returns

The total decompressed data block size.

3.3.2.4.4 uint32_t lz4_decompressBlock (const lz4_decompressBlockParams * *params,* lz4_status * *status*) Decompress a single LZ4 block without framing to a block of data.

Decompress a LZ4 block to an uncompressed data block. This API contains a single parameter to enable checking of the block checksum. When enabled the xxHash algorithm is used to verify the checksum. While not required it is the application programmers responsibility to determine if validating the checksum is necessary. See the lz4_decompressBlockParams structure documentation for more details about the available parameters.

Parameters

params	Pointer to the LZ4 decompression block parameter structure.
status	Pointer to a LZ4 status that will contain the result status.

Returns

The total decompressed data block size.

Referenced by lz4_decompress().

3.3.2.4.5 uint32_t lz4_getContentSize (const uint8_t * *src,* lz4_status * *status*) Get the content size of a compressed LZ4 frame.

Get the size of the original uncompressed data block if it is present. The content size is an optional parameter when compressing but it is recommended to enable it and verify the uncompressed data block fits in the allocated buffer before decompressing.

Parameters

src	Pointer to the LZ4 frame.
status	Pointer to a LZ4 status that will contain the result status.

Returns

The total decompressed content size. The content size is stored as 64-bit but will be truncated to 32-bit.

3.3.2.4.6 uint32_t lz4_stream_decompressBlock (lz4_stream_decompressBlockState * *state,* const void * *data,* uint16_t *length,* lz4_status * *status*) Decompress a single LZ4 block as a stream of data.

Continue decompression using a stream of data blocks. The streaming API's can be used when data is sent in chunks such as over-the-air or wired serial communication and removes the need to buffer then entire compressed data before running decompression, reducing total system memory used.

The lz4_stream_decompressBlockInit() function must first be called to initialize the state before decompressing data.

Parameters

state	Pointer to the LZ4 decompression stream state.
data	Pointer to a block of data to continue decompression with.
length	Length of data block in bytes.
status	Pointer to a LZ4 status that will contain the result status.

Returns

The current length of decompressed data.

3.3.2.4.7 void lz4_stream_decompressBlockInit (const lz4_stream_decompressBlockParams * *params*, lz4_stream_decompressBlockState * *state*, lz4_status * *status*) Initialize LZ4 stream decompression.

Initialize LZ4 decompression using a stream of data blocks. The streaming API's can be used when data is sent in chunks such as over-the-air or wired serial communication and removes the need to buffer then entire compressed data before running decompression, reducing total system memory used.

This function must be first called to initialize the state before calling lz4_stream_decompressBlock() to decompress data.

Parameters

params	Pointer to the LZ4 decompression strean parameter structure.
state	Pointer to the LZ4 decompression stream state.
status	Pointer to a LZ4 status that will contain the result status.

Returns

none

3.3.3 xxHash

Functions

- uint32_t xxhash_compute (const void *src, uint32_t length, uint32_t seed)

3.3.3.1 Detailed Description

The following is a reference of the xxHash API's that can be used to calculate the xxHash used in the LZ4 file format.

3.3.3.2 Function Documentation

3.3.3.2.1 uint32_t xxhash_compute (const void * *src*, uint32_t *length*, uint32_t *seed*) Compute xxhash checksum for a block of data.

Used to compute and verify checksums for the LZ4 frame and block format. This function has been optimized for MSP430 but it not as efficient as hardware CRC16 or CRC32 calculation.

Parameters

src	Pointer to the data block.
length	Length of data block.
seed	Initialization value.

Returns

The computed xxhash checksum.

Referenced by lz4_compress(), lz4_compressBlock(), lz4_decompress(), and lz4_decompressBlock().

3.4 Examples

Examples Overview 48
Compress Text to LZ4 File 48
Compress Text with CRC16 Checksum 48
Decompress Data Stream 48

3.4.1 Examples Overview

These examples demonstrate how to use the LZ4 compression utility to compress binary data.

3.4.2 Compress Text to LZ4 File

This example demonstrates how to compress a block of text into a valid LZ4 file with framing. The compressed data can be saved from memory as a raw binary file and recovered on the PC using the LZ4 command line utility.

This example is configured with a hash size of 4096 and includes both the content size and a content checksum to verify the original file contents. With these settings the LZ4 compression achieves a 1.51 compression ratio (3283 bytes compressed to 2169 bytes).

3.4.3 Compress Text with CRC16 Checksum

This example demonstrates how to compress a block of text into a custom frame with CRC16 checksum. The advantage is a significantly faster checksum calculation using the hardware CRC module compared to the software xxHash implementation.

```
//    +------------------+----------------+----------------------+
//    | 32-bit block size | compressed data | 16-bit CRC16 checksum |
//    +------------------+----------------+----------------------+
```

3.4.4 Decompress Data Stream

This example demonstrates how to decompress LZ4 data in chunks of compressed data. The advantage of the stream decompress functions is less data usage due to only needing to buffer small chunks, one at a time. The performance is slightly worse than the non-streaming API's.

3.5 LZ4 Benchmarks

LZ4 Compression Ratio 49
LZ4 Compression Speed 51

3.5.1 LZ4 Compression Ratio

3.5.1.1 Canterbury Corpus

The Canterbury Corpus is a set of small sized files that represent a wide range of data formats and is used to compare lossless compression algorithm performance. Applications can compare typical data sets with the corpus files to determine a rough estimate of compression performance. Since the benchmark does not measure compression speed the LZ4 compression utility has been compiled without modifications using GCC and run on a Windows PC to obtain the following results. The results are comparable to the original LZ4 implementation with the default command line options (fast compression).

File	Category	Size (bytes)	Ratio (HL2S 10)	Ratio (HL2S 12)	Ratio (HL2S 14)
`alice29.txt`	English text	152089	1.3079	1.5266	1.6782
`asyoulik.txt`	Shakespeare	125179	1.2684	1.4457	1.585
`cp.html`	HTML source	24603	1.6461	1.9097	2.0294
`fields.c`	C source	11150	1.9161	2.0614	2.0896
`grammar.lsp`	LISP source	3721	1.8512	1.9062	1.9141
`kennedy.xls`	Excel Spreadsheet	1029744	2.3826	2.5877	2.7481
`lcet10.txt`	Technical writing	426754	1.3491	1.6052	1.7899
`plrabn12.txt`	Poetry	481861	1.1735	1.3459	1.5018
`ptt5`	CCITT test set	513216	5.6318	5.8115	5.9357
`sum`	SPARC Executable	38240	1.6961	1.9148	1.9827
`xargs.1`	GNU manual page	4227	1.4526	1.5326	1.5632
Total		**2810784**	**1.8572**	**2.0987**	**2.2825**

Table 3.1: Canterbury Corpus benchmark LZ4 performance where HL2S is the hashLog2Size setting.

The benchmark files, additional details and results for other compression algorithms can be found on the Canterbury Corpus webpage (http://corpus.canterbury.ac.nz/).

3.5.1.2 Silesia Corpus

The Silesia Corpus is a set of large size files that represent a wide range of data formats and is used to compare lossless compression algorithm performance. While large file sizes are not typical in embedded applications they can still be used to compare typical data sets with the corpus files to determine a rough estimate of compression performance. Since the benchmark does not measure compression speed the LZ4 compression utility has been compiled without modifications using GCC and run on a Windows PC to obtain the following results. The results are comparable to the original LZ4 implementation with the default command line options (fast compression).

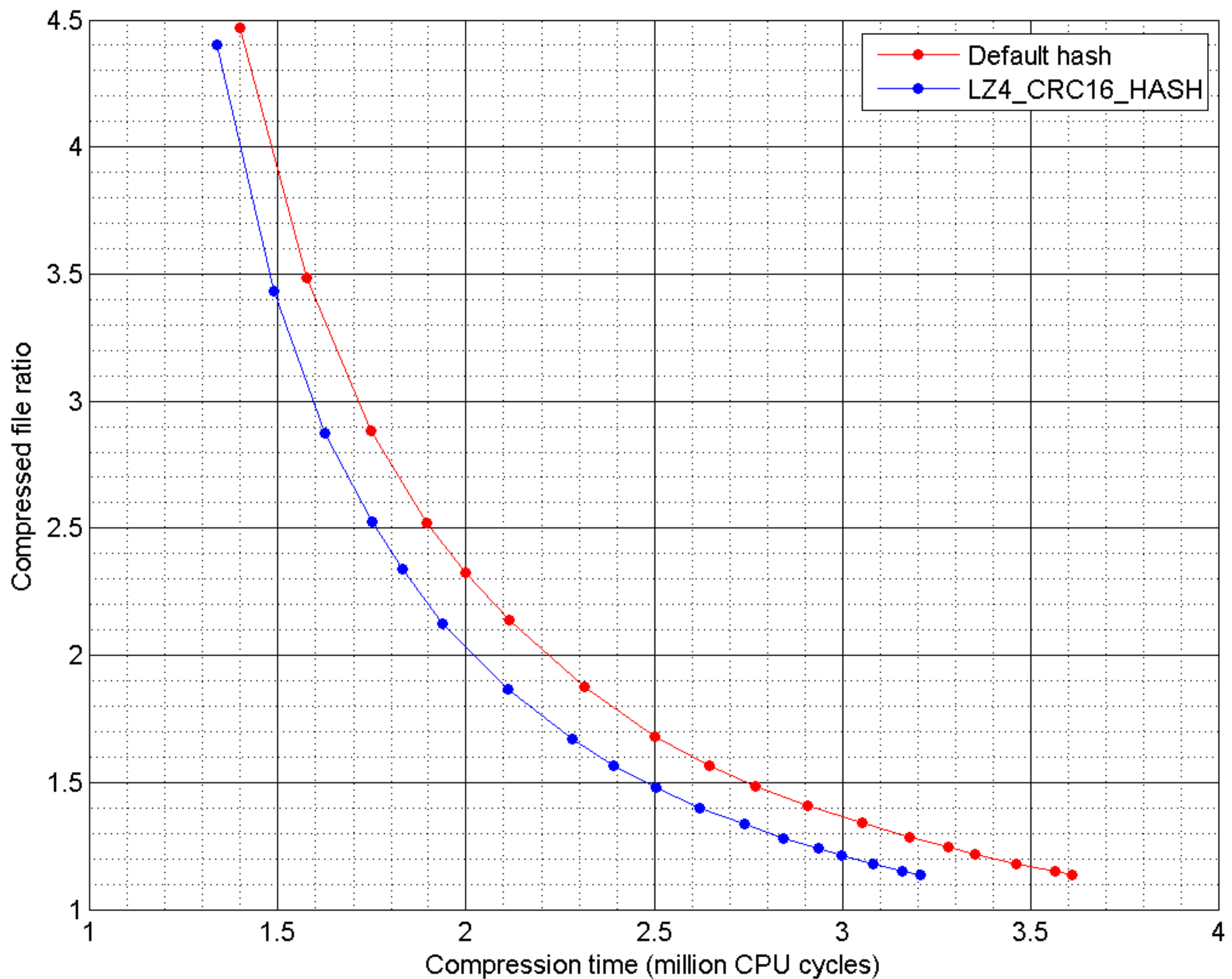
File	Category	Size (bytes)	Ratio (HL2S 10)	Ratio (HL2S 12)	Ratio (HL2S 14)
`dickens`	English text	10192446	1.2268	1.4259	1.5882
`mozilla`	Executable	51220480	1.7236	1.8536	1.9501
`mr`	Medical image	9970564	1.5728	1.6653	1.7521
`nci`	Database	33553445	4.5649	5.357	5.7073
`ooffice`	Executable	6152192	1.2934	1.3811	1.466
`osdb`	Database	10085684	1.1354	1.4785	1.965
`reymont`	Polish pdf	6627202	1.5583	1.7694	1.8747
`samba`	Source code	21606400	2.1857	2.5442	2.7149
`sao`	Binary data	7251944	1.0208	1.0538	1.0945
`webster`	HTML	41458703	1.5805	1.8459	2.0475
`xml`	HTML	5345280	3.2759	3.8561	4.1454
`x-ray`	Medical image	8474240	1.0036	1.0115	1.0441
total		**211938580**	**1.7244**	**1.9328**	**2.0910**

Table 3.2: Silesia Corpus benchmark LZ4 performance where HL2S is the hashLog2Size setting.

The benchmark files, additional details and results for other compression algorithms can be found on the Silesia Corpus webpage (http://sun.aei.polsl.pl/ sdeor/index.php?page=silesia).

3.5.2 LZ4 Compression Speed

The speed at which data can be compressed is directly related to the compression ratio of the data. Data that can be compressed at a higher ratio can be compressed faster because there are more matched bytes and fewer literals that must be hashed into the table. The plot below shows a synthetic benchmark created to compare the default multiply based hash function using the MPY32 peripheral with the CRC16 implementation. The original file being compressed is 32KB and a hash table size of 4096 is used.



As expected the hardware based CRC16 hash function is more efficient and results in faster compression speeds. As the compression ratio decreases the advantage becomes even more significant. This is because lower compression ratio results in less matches and more calls to the hashing function, decreasing the overall compression speed.

4 Random Number Generator (RNG)

Introduction 53
API Reference 54
Examples 56

4.1 Introduction

The random number generator (RNG) utility implements a counter mode deterministic random byte generator (CTR-DRBG) according to the NIST SP 800-90A Rev 1 specification. Random numbers are generated using seed information stored in the TLV tables unique to each device and generate 128-bits at a time (16-bytes). For more information on the implementation and test data refer to the SLAA725 application note linked below.

Random Number Generation Using the MSP430FR59xx/69xx

4.2 API Reference

API Overview 54
RNG 54

4.2.1 API Overview

The RNG utility includes a single API that can be used to generate random bytes using the CTR-DRBG methodology specified in NIST SP 800-90A Rev 1.

4.2.2 RNG

Macros

- #define RNG_KEYLEN

Functions

- uint16_t rng_generateBytes (uint8_t *dst, uint16_t length)

4.2.2.1 Detailed Description

The following is a reference of all API's available for the application to use.

4.2.2.2 Function Documentation

4.2.2.2.1 uint16_t rng_generateBytes (uint8_t * *dst,* uint16_t *length*) Generate random bytes and store to destination array.

Generates the requested number of random bytes using the CTR-DRBG methodology and the AES-128 block cipher algorithm according to Section 9.3.1 and 10.2.1.5.1 of NIST SP 800-90Ar1. The length parameter must be a multiple of RNG_KEYLEN, if it is not the length will be rounded down to the closest multiple and that many bytes will be generated and returned.

Note: Reseed, prediction resistance and additional inputs are not supported. Note: The security strength is fixed at 128-bit.

Parameters

dst	Pointer to the destination array to store generated bytes.
length	Number of bytes requested by the user, must be a multiple of RNG_KEYLEN bytes.

Returns

Length of random bytes that were generated.

4.3 Examples

Examples Overview 56
Generate Random Data 56
Generate Random Data to CSV File 56

4.3.1 Examples Overview

These examples demonstrate how to use the RNG utility to generate random data.

4.3.2 Generate Random Data

This example demonstrates how to generate random 8-bit and 32-bit data. The random bytes can be used for a variety of applications including cryptography and tamper detection.

This example generates data with length 64-bytes but data can be generated in any multiple of RNG_KEYLEN (16) bytes.

4.3.3 Generate Random Data to CSV File

This example demonstrates how to generate random data and write it to a CSV (comma-separated values) file. The generated CSV file can be used to analyze the randomness of generated bytes. For example the data can be read and plotted to a histogram in MATLAB with the following code:

```
//  data = csvread('rng_data.csv',1,1);
//  hist(data);
```

This example generates 2^14 random data bytes and takes approximately 4 minutes to write all values to the CSV file due to the limitations of file IO on MSP devices (using the debug stack and breakpoints to read/write data streams).

5 Non-Volatile Storage (NVS)

Introduction 57
API Reference 59
Examples 71

5.1 Introduction

The non-volatile storage (NVS) library makes handling of non-volatile data easy and robust against intermittent power loss or asynchronous device resets. MSP430 devices with FRAM non-volatile memory guarantee 16-bit writes in all scenarios however more often than not the data being stored is a larger data type or a structure containing multiple types. That means an unintended reset or power loss while data is written to the non-volatile memory will can result in partial data write and corrupted entries. To keep data storage constant, the non-volatile storage library contains functions that store data in a way that is guaranteed to recover the last valid entry without data corruption.

5.2 Features

The NVS library provides the following features:

- Recovery of latest valid entry
- Storage of any length data type or structure
- Application defined storage location
  - Persistent FRAM
  - Information memory
- Memory protection via MPU or SYS modules
- CRC protection

5.3 Storage Containers

The following storage containers are available in the NVS library.

5.3.1 Data Storage

The data storage is intended for single data structures. This could be device configurations, operating modes or operating counters like odometers. The data structures can be as simple as a single byte, or complex components like arrays or structs. Internally a double buffering scheme is used to be able to recover a known last state.

5.3.2 Log Storage

The log storage is intended for data logs. The corresponding functions allow adding data entries to the log and retrieve the data to a later point in time. The number of log entries has to be known ahead of time and the memory has to be reserved to hold the full log. Once the log is full, no further data can be added, unless the log is reset.

5.3.3 Ring Storage

The ring storage is intended for endless logging in a ring buffer configuration. For a ring buffer of n entries, only the last n-1 entries can be retrieved from the log.

5.4 Memory Allocation

The NVS library requires the application to allocated storage before initializing the storage container. The storage can be located in FRAM or information memory and the length can be calculated using the included preprocessor macros.

5.4.1 FRAM

The code snippet below demonstrate how to correctly allocate memory in FRAM for both CCS and IAR.

```
#if defined(__TI_COMPILER_VERSION__)
#pragma PERSISTENT(nvsStorage)
#elif defined(__IAR_SYSTEMS_ICC__)
__persistent
#endif
uint8_t nvsStorage[NVS_LOG_STORAGE_SIZE(SIZE, ENTRIES)] = {0};
```

5.4.2 Information Memory

The code snippet below demonstrate how to correctly allocate memory in INFOA for both CCS and IAR.

```
#if defined(__TI_COMPILER_VERSION__)
#pragma DATA_SECTION(nvsStorage, ".infoA")
#elif defined(__IAR_SYSTEMS_ICC__)
#pragma location="INFOA"
__no_init
#endif
uint8_t nvsStorage[NVS_LOG_STORAGE_SIZE(SIZE, ENTRIES)] = {0};
```

5.5 API Reference

NVS 59
NVS Data 59
NVS Log 63
NVS Ring 66
NVS Support 69

5.5.1 NVS

Enumerations

- enum nvs_status {
  NVS_OK, NVS_NOK, NVS_INDEX_OUT_OF_BOUND, NVS_CRC_ERROR,
  NVS_EMPTY, NVS_FULL }

5.5.1.1 Detailed Description

The following are shared API's between the three NVS storage containers.

5.5.1.2 Enumeration Type Documentation

5.5.1.2.1 enum nvs_status NVS return code and status information.

Enumerator

NVS_OK Successful operation.
NVS_NOK NVS storage format is corrupted.
NVS_INDEX_OUT_OF_BOUND Index is out of bounds.
NVS_CRC_ERROR Data checksum in incorrect.
NVS_EMPTY NVS storage is empty.
NVS_FULL NVS storage is full.

5.5.2 NVS Data

Data Structures

- struct nvs_data_header

Macros

- #define NVS_DATA_STORAGE_SIZE(size)
- #define NVS_DATA_TOKEN

Typedefs

- typedef void ∗ nvs_data_handle
- typedef struct nvs_data_header nvs_data_header

Enumerations

- enum nvs_data_status { NVS_DATA_INIT, NVS_DATA_1, NVS_DATA_2 }

Functions

- nvs_status nvs_data_commit (nvs_data_handle handle, void ∗data)
- nvs_data_handle nvs_data_init (uint8_t ∗storage, uint16_t size)
- nvs_status nvs_data_restore (nvs_data_handle handle, void ∗data)

5.5.2.1 Detailed Description

The following are types, macros and functions available for the data storage container.

5.5.2.2 Data Structure Documentation

5.5.2.2.1 struct nvs_data_header

NVS header for a non-volatile data storage container.

Data Fields

uint16_t	crc1	CRC for storage 1.
uint16_t	crc2	CRC for storage 2.
uint16_t	size	Size of data entry in byte.
nvs_data_status	status	Storage status.
uint16_t	token	Identifier token.

5.5.2.3 Enumeration Type Documentation

5.5.2.3.1 enum nvs_data_status

NVS data status flags.

Enumerator

NVS_DATA_INIT Successful operation.

NVS_DATA_1 Storage 1 contains the latest data.

NVS_DATA_2 Storage 2 contains the latest data.

5.5.2.4 Function Documentation

5.5.2.4.1 nvs_status nvs_data_commit (nvs_data_handle *handle,* void ∗ *data*)

Commit a data entry to the non-volatile storage container.

This function copies the data to the storage container. For integrity checks the CRC of the data is calculated and stored in the container as well.

Parameters

handle	NVS data container handle.
data	Pointer to a data structure that holds the data to be added to the storage container.

Returns

Status of the NVS operation.

5.5.2.4.2 nvs_data_handle nvs_data_init (uint8_t ∗ *storage,* uint16_t *size*) Initialize non-volatile data storage container.

This function checks for an existing non-volatile data container at the given location. If it finds an existing container, it will match the properties of the container and verify the constancy of the container. A CRC check of the data is performed and if everything is match the function will return without any modification to the storage container. In case of a failing CRC an incomplete storage commit is assumed and the alternate storage buffer is checked. If the CRC check is OK, the status is updated and will point to the alternate storage buffer. Only when no container is found, or the properties of the container have changed, or no matching CRC was found, then the container will be initialized.

Example non-volatile-storage space and function call:

- unit8_t nvs_data_container[NVS_DATA_STORAGE_SIZE(sizeof(DATA))];
- nvs_data_init(nvs_data_container, sizeof(DATA));

Parameters

storage	Pointer to NVS data storage with size calculated using NVS_DATA_STORAGE_SIZE.
size	Size of data structure element that is being stored.

Returns

NVS data container handle.

5.5.2.4.3 nvs_status nvs_data_restore (nvs_data_handle *handle,* void ∗ *data*) Restore a data entry from the non-volatile data storage container.

This function does restore a data entry by copying the most recent data from the storage container to the data location. The size of the data is defined during the nvs_data_init function call and is fixed for every nvs_data_container. After the data has been copied, the CRC of the data is calculated and compared against the stored CRC value. The result is reflected in the return value.

Parameters

handle	NVS data container handle.
data	Pointer to a data structure that will hold the data after the restore operation.

Returns

Status of the NVS operation.

5.5.3 NVS Log

Data Structures

- struct nvs_log_header

Macros

- #define NVS_LOG_STORAGE_SIZE(size, num)
- #define NVS_LOG_TOKEN

Typedefs

- typedef void * nvs_log_handle
- typedef struct nvs_log_header nvs_log_header

Functions

- nvs_status nvs_log_add (nvs_log_handle handle, void *data)
- uint16_t nvs_log_entries (nvs_log_handle handle)
- bool nvs_log_full (nvs_log_handle handle)
- nvs_log_handle nvs_log_init (uint8_t *storage, uint16_t size, uint16_t length)
- uint16_t nvs_log_max (nvs_log_handle handle)
- nvs_status nvs_log_reset (nvs_log_handle handle)
- nvs_status nvs_log_retrieve (nvs_log_handle handle, void *data, uint16_t index)

5.5.3.1 Detailed Description

The following are types, macros and functions available for the log storage container.

5.5.3.2 Data Structure Documentation

5.5.3.2.1 struct nvs_log_header

NVS type definition for a non volatile LOG storage container.

Data Fields

uint16_t	index	Index of last log entry.
uint16_t	length	Maximum number of entries in log storage.
uint16_t	size	Size of data entry in bytes.
uint16_t	token	Identifier token.

5.5.3.3 Macro Definition Documentation

5.5.3.3.1 #define NVS_LOG_STORAGE_SIZE(*size*, *num*)

Calculate the NVS log storage size from structure size and number of elements.

5.5.3.4 Function Documentation

5.5.3.4.1 nvs_status nvs_log_add (nvs_log_handle *handle*, void * *data*)

Adds a data entry to the non-volatile LOG storage container.

This function copies the data to the storage container. For integrity checks the CRC of the data is calculated and stored in the container as well.

Parameters

handle	NVS log container handle.
data	Pointer to data structure to add to the storage container.

Returns

Status of the NVS operation.

5.5.3.4.2 uint16_t nvs_log_entries (nvs_log_handle *handle*)

Return the number of valid entries in the log container.

This function will return the number of valid entries of the log container.

Parameters

handle	NVS log container handle.

Returns

Number of valid entries.

5.5.3.4.3 bool nvs_log_full (nvs_log_handle *handle*)

Check whether the log storage container is full.

This function does check whether the log storage container is full.

Parameters

handle	NVS log container handle.

Returns

True if NVS log is full.

5.5.3.4.4 nvs_log_handle nvs_log_init (uint8_t * *storage*, uint16_t *size*, uint16_t *length*)

Initialize non-volatile data LOG storage container.

This function checks for an existing non-volatile log container at the given location. If it finds an existing container, it will match the properties of the container and verify the constancy of the container. A CRC check of the most recent data is performed and if everything is match the function will return without any modification to the storage container. In case of a failing CRC an incomplete storage operation is assumed and the log container is verified for consistency from the beginning. The last good known data/CRC match is used as the new end of log index.

Only when no container is found, or the properties of the container have changed, or no matching CRC was found, then the container will be initialized.

Parameters

storage	Pointer to NVS data storage with size calculated using NVS_LOG_STORAGE_SIZE.
size	Size of data structure element that is being stored.
length	Maximum number of entries in the log storage.

Returns

NVS log container handle

5.5.3.4.5 uint16_t nvs_log_max (nvs_log_handle *handle*) Return the max number of entries for this log container.

This function will return the maximum number of allowed entries for the given log container.

Parameters

handle	NVS log container handle.

Returns

Maximum number of allowed entries in log container.

5.5.3.4.6 nvs_status nvs_log_reset (nvs_log_handle *handle*) Reset (clear) non-volatile data LOG storage container.

This function will reset/clear the non-volatile log container. All CRC values will be invalidated, so no data can be recovered.

Parameters

handle	NVS log container handle.

Returns

Status of the NVS operation.

5.5.3.4.7 nvs_status nvs_log_retrieve (nvs_log_handle *handle,* void * *data,* uint16_t *index*) Retrieve a specific data entry from the non-volatile log storage container.

This function does retrieve a specific data entry by copying the data with the given index from the storage container to the data location. The size of the data is defined during the nvs_log_init function call and is fixed for every nvs_log_container entry. After the data has been copied, the CRC of the data is calculated and compared against the stored CRC value. The result is reflected in the return value.

Parameters

handle	NVS log container handle.
data	Pointer to data structure to populate with retrieved data.
index	Index of data to retrieve.

Returns

Status of the NVS operation.

5.5.4 NVS Ring

Data Structures

- struct nvs_ring_header

Macros

- #define NVS_RING_STORAGE_SIZE(size, num)
- #define NVS_RING_TOKEN

Typedefs

- typedef void * nvs_ring_handle
- typedef struct nvs_ring_header nvs_ring_header

Functions

- nvs_status nvs_ring_add (nvs_ring_handle handle, void *data)
- uint16_t nvs_ring_entries (nvs_log_handle handle)
- bool nvs_ring_full (nvs_ring_handle handle)
- nvs_ring_handle nvs_ring_init (uint8_t *storage, uint16_t size, uint16_t length)
- uint16_t nvs_ring_max (nvs_ring_handle handle)
- nvs_status nvs_ring_reset (nvs_ring_handle handle)
- nvs_status nvs_ring_retrieve (nvs_ring_handle handle, void *data, uint16_t index)

5.5.4.1 Detailed Description

The following are types, macros and functions available for the ring storage container.

5.5.4.2 Data Structure Documentation

5.5.4.2.1 struct nvs_ring_header

NVS type definition for a non volatile RING storage container.

Data Fields

uint16_t	first	Index of first ring entry.
uint16_t	last	Index of last ring entry.
uint16_t	length	Maximum number of entries in ring storage.
uint16_t	size	Size of data entry in bytes.
uint16_t	token	Identifier token.

5.5.4.3 Macro Definition Documentation

5.5.4.3.1 #define NVS_RING_STORAGE_SIZE(*size*, *num*) Calculate the NVS ring storage size from structure size and number of elements.

5.5.4.4 Function Documentation

5.5.4.4.1 nvs_status nvs_ring_add (nvs_ring_handle *handle*, void * *data*) Adds a data entry to the non-volatile RING storage container.

This function copies the data to the storage container. For integrity checks the CRC of the data is calculated and stored in the container as well.

Parameters

handle	NVS ring container handle.
data	Pointer to data structure to add to the storage container.

Returns

Status of the NVS operation.

5.5.4.4.2 uint16_t nvs_ring_entries (nvs_log_handle *handle*) Return the number of valid entries in the ring container.

This function will return the number of valid entries of the ring container.

Parameters

handle	NVS ring container handle.

Returns

Number of valid entries.

Referenced by nvs_ring_retrieve().

5.5.4.4.3 bool nvs_ring_full (nvs_ring_handle *handle*) Check whether the ring storage container is full.

This function does check whether the ring storage container is full.

Parameters

handle	NVS ring container handle.

Returns

True if NVS ring is full.

5.5.4.4.4 nvs_ring_handle nvs_ring_init (uint8_t * *storage*, uint16_t *size*, uint16_t *length*) Initialize non-volatile data RING storage container.

This function checks for an existing non-volatile ring container at the given location. If it finds an existing container, it will match the properties of the container and verify the constancy of the container. A CRC check of the most recent data is performed and if everything is match the function will return without any modification to the storage container. In case of a failing CRC an incomplete storage operation is assumed and the ring container is completely analyzed to identify the oldest and most recent data entry. The ring container is then updated to contain the updated first/last information.

Only when no container is found, or the properties of the container have changed, or no matching CRC was found, then the container will be initialized with an empty ring buffer.

Parameters

storage	Pointer to NVS data storage with size calculated using NVS_RING_STORAGE_SIZE.
size	Size of data structure element that is being stored.
length	Maximum number of entries in the ring storage.

Returns

NVS ring container handle

5.5.4.4.5 uint16_t nvs_ring_max (nvs_ring_handle *handle*) Return the max number of entries for this ring container.

This function will return the maximum number of allowed entries for the given ring container.

Parameters

handle	NVS ring container handle.

Returns

Maximum number of allowed entries in ring container.

5.5.4.4.6 nvs_status nvs_ring_reset (nvs_ring_handle *handle*) Reset (clear) non-volatile data RING storage container.

This function will reset/clear the non-volatile ring container. All CRC values will be invalidated, so no data can be recovered.

Parameters

handle	NVS ring container handle.

Returns

Status of the NVS operation.

5.5.4.4.7 nvs_status nvs_ring_retrieve (nvs_ring_handle *handle,* void * *data,* uint16_t *index*) Retrieve a specific data entry from the non-volatile ring storage container.

This function does retrieve a specific data entry by copying the data with the given index from the storage container to the data location. An index of 1 does point to the oldest entry and every

increment will return one younger data entry. The size of the data is defined during the nvs_log_init function call and is fixed for every nvs_log_container entry. After the data has been copied, the CRC of the data is calculated and compared against the stored CRC value. The result is reflected in the return value.

Parameters

handle	NVS ring container handle.
data	Pointer to data structure to populate with retrieved data.
index	Index of data to retrieve.

Returns

Status of the NVS operation.

5.5.5 NVS Support

Functions

- uint16_t nvs_crc (void ∗data, uint16_t size)
- void nvs_lockFRAM (uint16_t state)
- uint16_t nvs_unlockFRAM (void)

5.5.5.1 Detailed Description

The following are support functions used within the NVS library.

5.5.5.2 Function Documentation

5.5.5.2.1 uint16_t nvs_crc (void ∗ *data*, uint16_t *size*)

Calculate a 16-bit CRC over a storage buffer in bytes.

Parameters

data	Pointer to data to calculate CRC on.
size	Length of data array.

Returns

none

Referenced by nvs_data_commit(), nvs_data_init(), nvs_data_restore(), nvs_log_add(), nvs_log_init(), nvs_log_retrieve(), nvs_ring_add(), nvs_ring_init(), and nvs_ring_retrieve().

5.5.5.2.2 void nvs_lockFRAM (uint16_t *state*) `[inline]`

Lock FRAM after writing.

Restore the previous FRAM protection state with the state returned from nvs_unlockFRAM().

Parameters

state	FRAM state returned from nvs_unlockFRAM().

Returns

none

Referenced by nvs_data_commit(), nvs_data_init(), nvs_log_add(), nvs_log_init(), nvs_log_reset(), nvs_ring_add(), nvs_ring_init(), and nvs_ring_reset().

5.5.5.2.3 uint16_t nvs_unlockFRAM (void) [inline] Unlock FRAM for writing.

Clear the FRAM program write protection bit and return the original status of the bit. The return of this function can be used to restore the previous FRAM protection state with nvs_lockFRAM().

Returns

FRAM state that can be passed into nvs_lockFRAM()

Referenced by nvs_data_commit(), nvs_data_init(), nvs_log_add(), nvs_log_init(), nvs_log_reset(), nvs_ring_add(), nvs_ring_init(), and nvs_ring_reset().

5.6 Examples

Examples Overview 71
Continuous counter 71
Store application configuration 71
Log structure to FRAM 71
Black box recorder 71

5.6.1 Examples Overview

These examples demonstrate how to use the NVS utility to store data in non-volatile FRAM or information memory.

5.6.2 Continuous Counter

This example demonstrates how to implement a continuous up counter that never looses its value regardless of asynchronous reset or power cycle events. Even a compile and program update does not disrupt the counter, as long as the NVS container stays the same.

5.6.3 Application Configuration

This example demonstrates how to store a complex structure in the non volatile memory. The library function will assure to always retrieve a complete set of variables regardless of asynchronous reset or power cycle events. Even a compile and program update does not disrupt the configuration as long as the NVS container stays the same.

5.6.4 Data Logger

This example demonstrates how to utilize the data logger functionality of the NVS library to log a structure containing timestamp and ADC measurement. The main program will log until NVS storage is full and then read back and print logged data.

5.6.5 Black Box Recorder

This example demonstrates how to utilize the NVS ring storage container to create a black box recorder with the most recent samples. The main program will log a specified number of entries and then read back and print the latest data.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

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