Application Report

High Endurance EEPROM Emulation Driver for TM4C129x Devices

Bob Crosby

ABSTRACT

The EEPROM module of the TM4C129x microcontroller family provides a hardware-based emulation of EEPROM using flash memory. This implementation provides a write-erase endurance of 500K cycles. This application report provides a software driver that is used on top of the EEPROM module hardware and can greatly extend the write-erase endurance.

Project collateral and source code discussed in this application can be downloaded from the following URL: http:/www.ti.com/lit/zip/spma078.

Table of Contents

Trademarks

1 Introduction

The FEE driver uses two physical sectors of the EEPROM flash, which is 16 blocks or 1024 bytes for each dataset. The TM4C129 devices support up to six datasets. The size of the dataset determines how many images of the data can be stored in the 1024 bytes. This will then determine the supported write-erase endurance specification. The use of two sectors will allow retaining a valid copy of the data in the case of power loss or reset while trying to program or erase. The relationship between dataset size and supported write-erase endurance is shown in Table 1-1.

Table 1-1. Dataset Size and Write-Erase Endurance

Dataset Maximum Size (Words)	Dataset Minimum Size (Words)	Supported Write-Erase Cycles	Number of Images
124	61	1M	2
60	39	2M	4
38	29	3M	6
28	22	4M	8
21	18	5M	10
17	15	6M	12
14	13	7M	14
12	11	8M	16
10	9	9M	18
8	8	10M	20
7	7	11M	22
6	6	12M	24
5	5	13M	26
4	4	16M	32
3	3	18M	36
2	2	21M	42
1	1	25M	50

1.1 Definitions

Block A 64-byte segment of the TM4C129 EEPROM module.

Dataset A group of data that is written to the EEPROM in a single call to the FeeWrite() function. The size of a dataset is pre-defined at compile time in a customizable header file.

EEPROM The flash emulated “electrically erasable programmable read-only memory” hardware module of the TM4C129 family of devices.

FEE A flash emulated EEPROM driver

Image An instance of a dataset stored in a pair of sectors. Only one image in the sector-pair is the most recent image and it holds the current data for that dataset.

Sector The smallest erasable set of flash cells in the EEPROM module. For the TM4C129 family, a sector contains 512 bytes. In the datasheet a sector is referred to as a “meta-block”.

Word Four bytes, the native integral size of the TM4C129 family.

2 Theory of Operation

2.1 Normal Use

The EEPROM of the TM4C129x device is composed of a bank of split gate flash. The flash cells have an intrinsic write-erase endurance of more than 100,000 cycles. The bank is implemented with 13 sectors. Twelve sectors are used for data and the thirteenth is used for the copy buffers. A sector is the smallest amount that can be erased at one time. Each of the twelve data sectors is broken into eight blocks. Each block holds 16 32-bit words (64 bytes). Within that block there are actually storage locations for seven images of the 16 words, with additional flash control bits to identify which of the seven images holds the valid data.

In normal operation, the EEPROM hardware state-machine keeps track of which of the seven images of a word are valid. When the program tries to update a word the eighth time, the word is initially written into the copy sector. Then the current value of each of the other 127 words ((16 words x 8 blocks) – 1) is also copied to the copy sector. The data sector is then erased, and the data in the copy sector is programmed into the first image of the data sector.

The copy sector, like the data sectors, has multiple images. There are 12 image locations in the copy sector. That means that before the thirteenth time a data sector must be erased, the copy sector must be erased to prepare more room. The hardware state-machine will initiate an erase of the copy sector the next time a data sector requires an erase. This results in a longer delay before the data can be stored in non-volatile memory.

2.2 Normal Endurance

Since each flash cell is capable of over 100,000 write-erase cycles, and there are seven image copies available for each word in a block, the expectation is that the word is capable of over 700,000 write-erase cycles. That is true, but other words in the sector have an impact on the total number of write-erase cycles that any flash cell sees. A write-erase cycle will occur on a flash cell each time it is changed from a zero to a one. That is, each time electrons are pulled off of the floating gate through the erase oxide. Take an extreme example of two words programmed into a data block. The first word is updated 700,000 times, the second word is not updated at all. Assume the least significant bit of both words is always 0. Because of the 700,000 writes, the sector containing that block will be erased 100,000 times. As expected, the word that was changed 700,000 times will have seen 100,000 write-erase cycles on the least significant bit in all seven images. Those bits will be nearing end of life. Perhaps unexpectedly, the least significant bit of the second word, which was never updated, will also have seen 100,000 write-erase cycle in just the first image. That bit too will be nearing end of life.

In typical applications, different words are updated at varying frequencies. Since the word most often updated, may not be the one updated the most often within a stretch of seven updates, a conservative limit of 500,000 cycles is specified for the most often updated word in the sector.

When large amounts of data are written in a single sweep, the full 600,000 write-erase cycle advantage of the wear leveling can be achieved. Adding software wear leveling, even greater number of write-erase cycles are possible.

Note

When doing sweep programming you get an effective 600,000 write-erase cycles instead of 700,000 write erase cycles. As soon as you write the first word of the sweep the eighth time, the sector is erased and the eighth value of the first word is programmed into the freshly erased location, along with the seventh version of all the other words in the sweep. Then when you update the second word of the sweep, it occupies the second image of that location. You only have five image locations left for the second location. When you write the first word of the sweep the fourteenth time, it resides in the seventh image. But when you write the fourteenth version of the second word, it initiates a sector copy. If you always program the words of the sweep in the same order, successive words will initiate the sector copy. You will have an effective six versions of the sweep data stored between each sector copy operation.

2.3 High Endurance Use

This project proposes a high endurance EEPROM model that combines software wear leveling with the hardware wear leveling for extended write-erase endurance. The plan is to use two physical sectors of the EEPROM flash, which is 16 blocks or 1024 bytes of the 6K bytes available. The use of two sectors will reduce the possibility of data loss when power is lost during programming or erase operations.

At the beginning of each dataset will be a control block that consists of the following four words:

- CycleCount1 (32-bit)
- CycleCount2 (32-bit)
- CycleCount3 (32-bit)
- CRC (32-bit)

2.4 Data Integrity

The most difficult part of a flash based EEPROM emulation driver is ensuring data integrity during unexpected loss of power or reset. While the hardware-based state-machine goes a long way to address this concern, it falls short as evidence by erratum MEM#07. Bits may be only partially erased or only partially programmed because the erase or program operation was cut short by power loss or reset. The actual programming time is much shorter than the erase time. (Programming is done with hot carrier injection, erase is done by Fowler-Nordheim tunneling.) Therefore, the most common issue with data integrity is with a partially erased sector. In this case, the sector may appear as totally erased, but some bits are on the threshold between 1 and 0. These bits may flip when being read. To avoid the partially erased sector, two sectors are used. The driver alternates between the sectors when it updates the data. After the data is written, the driver writes the CRC that was calculated from the buffer. The driver then runs a CRC calculation on the data in the EEPROM. Only if the CRC which was calculated on the buffer and then stored and retrieved from the EEPROM matches the CRC that was calculated from the EEPROM contents is the cycle count updated. If there is a mis-match, the write operation fails. The application may choose to rewrite the data. If there is an error indicated by the internal state-machine during a program or erase operation, this software will repeat the operation up to seven times. This is enough times to force a sector copy operation that would resolve corrupt data caused by erratum MEM#07. It is important that the writing of CycleCount3 not trigger a sector copy, as this operation is done after the data was validated. Because of this the seven retries are disabled during that write. A failure writing CycleCount3 will result in the write operation failing.

If the data in a dataset is programmed in the same order every time, the actual write operation which triggers the sector copy will progress down the line from the first word, to the second and eventually the cycle count. To avoid this, we use three copies of the counter. We write the first two instances of the counter before updating an image and update the third instance only after all data and the CRC have been updated. We alternate writing the first or the second instance of the counter first. This way, it will always be a write to one of the first two counters that initiates the sector copy operation.

If there is a power loss or reset during the programming process, that image of data is lost. The previous most recent image of data will be used.

In the read operation the driver calculates the CRC of the data read and compares it to the value stored in the control block. If there is a mismatch, the driver searches for the newest “old” copy of the data that has a good CRC value and reads that data with a return result of *FEE_OLD*.

2.5 CRC Module

The hardware CRC module is used by the FEE driver to validate that the contents of the emulated EEPROM memory are the same as what was intended. The CRC module is used in the functions FeeWrite(), FeeRead(), FeeInit(), FeeCheckDatasetValid() and FeeMainFunction(). The configuration, use and reading of the module are completely contained within each function call. The hardware CRC module may be used with different settings if the state of the module is not required to be maintained across a call to a FEE function.

If the hardware CRC module is to be used in a manner that calls to the Fee functions occur while calculating with the CRC module, the configuration used by the application must be defined in the HEeepromConfig.h file. This causes the FEE driver to save the contents of the CRCSEED and restore if after completion. It will also restore the CRCCTRL register to the settings defined in the HEeepromConfig.h file. Since the Fee driver cannot restore the contents of the Post Processing Result (CRCRSLTPP) register, if a FEE driver function is called after the last write to the CRC Data Input (CRCDIN) register and before reading the post processing result register, the contents of the post processing result register will be corrupt. In this case, the CRC Seed (CRCSEED) register should be read instead and the post processing performed by the CPU.

If the CRC module is to be used in an interrupt routine and the FEE status is not *FEE_IDLE*, the contents of the CRCSEED register should be saved before using the module. After using the module, the configuration should be set back to (CRC_CFG_TYPE_P4C11DB7 | CRC_CFG_SIZE_32BIT) and the saved value restored as the new seed.

3 Software Description

3.1 Environment

The software was built and tested using version 20.2.4 LTS of the Texas Instruments ARM C – Compiler and version 2.2.0.295 of the TivaWare driver library.

3.2 Files

The high endurance EEPROM emulator consists of one source file and two header files.

fee.c	Source file, should not be modified.
fee.h	Header file, should not be modified.
feeConfig.h	Header file, this file should be modified to set the number of datasets and their size, the starting EEPROM block and if interrupts are to be used.

3.3 Functions

3.3.1 FeeInit

This function initializes the high endurance EEPROM functions. It calls EEPROMInit() which resets and initializes the EEPROM module. It identifies the most recent image for each of the datasets. If any dataset does not have a valid image, it returns FEE_NOT_OK. If all datasets have a valid image, but one or more of the datasets had to choose an older image because the image with the highest CycleCount3 value was corrupt, then the function returns FEE_OLD and sets the module state to FEE_IDLE. Otherwise it returns *FEE_OK* and sets the module state from FEE_UINIT to FEE_IDLE. If the function returns FEE_NOT_OK, the program should check all of the datasets by calling FeeCheckDatasetValid() to identify datasets without any valid data. Calling FeeFormat() for that dataset will reformat the dataset, but the old data will be lost.

```
tFeeReturn FeeInit(void);
```

Parameters in: None

Return: enumeration tFeeReturn

FEE_OK

FEE_OLD

FEE_NOT_OK

3.3.2 FeeCheckDatasetValid

This function checks a dataset for a valid image. To be valid,

- all three counters of the image must match
- the counter value should correlate to the sector, odd or even
- the stored CRC must match the CRC of the data

```
tFeeReturn FeeCheckDatasetValid(uint8_t ui8Dataset);
```

Parameters in: The index of the dataset to be formated

Return: enumeration tFeeReturn

FEE_OK

FEE_OLD

FEE_NOT_OK

3.3.3 FeeFormat

This function forces an initialization of one dataset of the FEE. Any data in that dataset is lost. The dataset will have one valid image with the data all set to 0xFFFFFFFF. It sets the module state to *FEE_BUSY* and the initialization is finished by calls to FeeMainFunction(). Once reformatted, the counters will be reset. At the end of formatting the dataset, FeeMainFunction() will set the FEE status from *FEE_UNINIT* to *FEE_IDLE* only if all datasets are now valid.

```
void FeeFormat(uint8_t ui8Dataset);
```

Parameters in: The index of the dataset to be formated

Return: none

3.3.4 FeeRead

This function will read the most current version of the data. The buffer must be large enough to hold the dataset. This function requires the state-machine to be idle. A call to FeeRead() when the state-machine is not idle will return *FEE_NOT_OK*. Reading a dataset whose most recent image was corrupted will return *FEE_OLD.* Reads are performed synchronously and do not require calls to FeeMainFunction() to complete.

```
tFeeReturn FeeRead(uint8_t ui8Dataset, uint32_t* pui32DataBuffer);
```

Parameters in: ui8Dataset: The index of the dataset to be

read

pui32DataBuffer: Pointer to a databuffer to in which to store the data that is read

Return: enumeration tFeeReturn
FEE_OK: The read was completed without issue.

FEE_OLD: The most current version of the data was corrupt so an older version was returned.

FEE_NOT_OK: The read was not done because the driver was not idle or the dataset did not hold any valid data.

3.3.5 FeeWrite

This function starts the process of writing new data into the EEPROM. The buffer must be held constant until the write is complete. This is determined by calls to FeeGetStatus() with a return of *FEE_IDLE*. The function FeeMainFunction() is used to complete the process of writing the new data and updating the control words.

```
tFeeReturn FeeWrite(uint8_t ui8Dataset, uint32_t* pui32DataBuffer);
```

Parameters in: ui8Dataset: The index of the dataset to be written.

pui32DataBuffer: Pointer to a databuffer which contains the data to be written

Return: enumeration tFeeReturn
FEE_OK: The write request was accepted.

FEE_NOT_OK: The write request was not accepted, the FEE status was not *FEE_IDLE*.

3.3.6 FeeGetStatus

This function returns the status of the software FEE state-machine.

```
tFeeStatus FeeGetStatus(void);
```

Parameters In: none

Return: Enumeration tFeeStatus
FEE_UNINIT: The FEE driver has not been successfully initialized.
FEE_IDLE: The FEE driver is currently idle.
FEE_BUSY: The FEE driver is currently busy.

3.3.7 FeeGetJobResults

This function returns the state of the last job request.

```
tFeeJobResult FeeGetJobResult(void);
```

Parameters In: none

Return: Enumeration tFeeJobResult

FEE_JOB_OK: The last job has finished successfully.
FEE_JOB_PENDING: The last job is waiting for execution or is currently being executed.
FEE_JOB_FAILED: The last job failed.

3.3.8 FeeGetDatasetCounter

This function returns the current counter value of the selected dataset. This function is to be called only after a successful call to FeeInit(). The counter value can be used as an indication of the number of times that a dataset has been written. The counter is reset if the dataset is reformatted.

```
uint32_t FeeGetDatasetCounter(uint8_t ui8Dataset);
```

Paramaters In: ui8Dataset: The index of the dataset whose counter is to be read.

Return: The current count value of that dataset.

3.3.9 FeeGetVersionInformation

This function returns a two-byte structure with a major and a minor version number.

```
tFeeVersion FeeGetVersionInformation(void);
```

Parameters In: none

Return: Structure tFeeVersion

```
Typedef struct
{
    uint8_t ui8MajorVersion;
    uint8_t ui8MinorVersion;
}tFeeVersion;
```

3.3.10 FeeMainFunction

This function must be called repeatedly to complete a FeeWrite() or FeeFormat() operation. Each call to FeeMainFunction() will only program one word, four bytes. An additional four words must be programmed for the control data and the CRC value. Calls to FeeMainFunction() while the state is *FEE_BUSY* will return immediately. This function can be repeatedly called by the user application code or by the flash interrupt service routine.

```
void FeeMainFunction(void);
```

Parameters In: none

Return: none

4 Execution Time

4.1 Read Time

The FeeRead() function is asynchronous so its execution time is dependent on the system clock frequency and the number of data values in the dataset. Table 4-1 shows typical execution time for FeeRead() with different size datasets at 120 MHz system clock.

Table 4-1. FeeRead() Execution Time

Dataset Size (Words)	Execution Time
4	6.2 µs
5	7.1 µs
6	8.0 µs
7	8.9 µs
8	9.8 µs

4.2 Write Time

The write time of a dataset is dependent on the number of erase cycles seen by the sector. The erase time increases after many write-erase cycles. The write time is shown in logic analyzer captures for each of three conditions.

4.2.1 Write Time When no Erase is Required

When no erase is required, the write time is fairly constant at about 290 µs per word. There are four control words for each dataset. The time to write a dataset with four words of data is shown in Figure 4-1 from markers A to D and is 2.4 ms. Half the time is for writing the control words, the other half for writing the data.

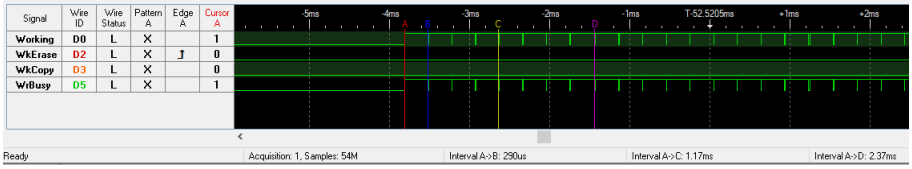


Figure 4-1. Program Time With no Erase

4.2.2 Write Time when Sector Copy is Required

When a sector copy operation is required the dataset write time is extended by the time required to copy all the data from the sector into a copy buffer, the time to erase the sector and the time to copy the data back into the newly erased sector. The erase time will increase as the device ages in terms of write-erase cycles. Figure 4-2 shows the copy and erase time of a device in mid-life. On this part, the copy, erase, copy back steps added about 73 ms to the write time of the dataset.

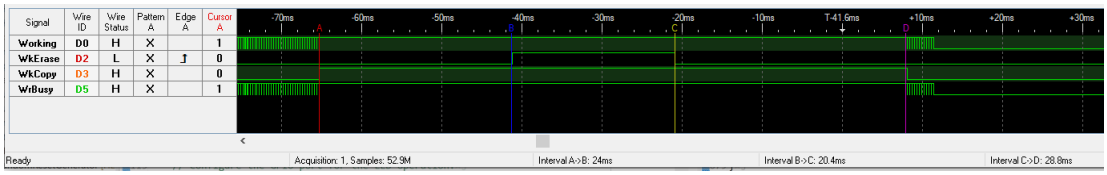


Figure 4-2. Program Time With Sector Erase

4.2.3 Write Time when Sector Copy is Required and Copy Sector is Full

When a sector copy operation is required and the copy sector is already full, the copy sector must first be erased before the copy operation can start. This is shown in Figure 4-3 below. On this part, erasing the copy sector added an additional 20 ms to the write time of the dataset.

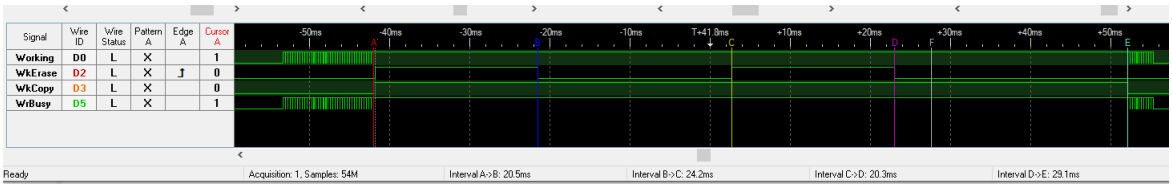


Figure 4-3. Program Time with Sector Erase and Copy Sector Erase

4.2.4 Write Time with Interrupts

As shown in Section 4.2.1, the time to write a four word dataset is about 2.4 ms. When in polling mode, the CPU is waiting the full 2.4 ms for the write operations to complete. Writing in interrupt mode takes the same time, but the CPU is able to perform other tasks. Figure 4-4 below shows the time to complete writing a four word dataset in interrupt mode. There are eight writes, four data words and four control words. The first write start is indicated by marker A in the figure below. After roughly 290 µs, the first write completes and an interrupt is generated. The interrupt service routine takes roughly 10 µs and starts the second write. The total CPU time required to write the four word dataset is roughly 90 µs out of the 2.4 ms. That inclues an initial 10 µs for the FeeWrite() function and 10 us for each of eight interrupts.

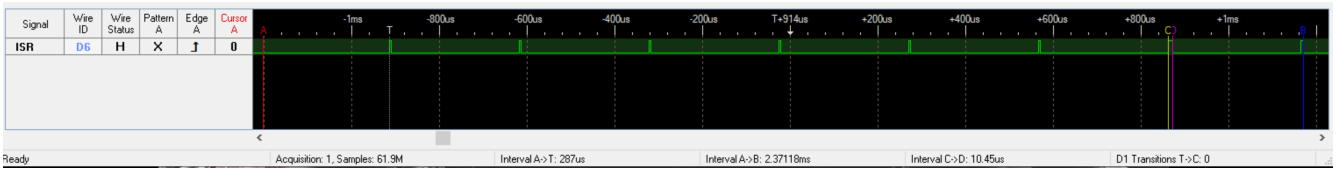


Figure 4-4. Interrupt Routine Execution Time

5 Examples

Two examples are provided with this application note. Both examples run on the EK-TM4C1294XL or EK-TM4C129EXL LaunchPad. They are similar but the first uses a polling method and the second uses interrupts.

There is a third project that is included in the collateral. The project EE_RamBasedErase can be loaded into RAM and executed to erase the contents of the TM4C129x device EEPROM. This will make the device EEPROM mimick the state of a new device.

5.1 Polling Fee Example

This example uses the FEE driver to maintain two counters. Upon reset, the count values are checked and the values are output on the virtual serial port attached to UART0. Then LEDs D1 and D2 are alternately illuminated while the status of user switches SW1 and SW2 are read. While SW1 is depressed, the counter in dataset zero will be incremented and the new value will be output on the serial port. Likewise, when SW2 is depressed, the counter in dataset one will be incremented and the new value will be output on the serial port.

In each dataset the counters are implemented with four copies of the count. Two copies in clear form, and two copies stored in one's complement. Each time a dataset is read, the CRC is checked and all four copies are checked to verify they contain the same count. Any errors are reported out the serial port.

5.2 Interrupt Fee Example

In both examples, when neither switch is depressed, the LEDs alternate at a rate of about ten times per second. This is accomplished with a simple software loop. When a switch is depressed, extra time is required to update the FEE dataset and output the value on the UART. This is usually not noticed by the casual observer. However, periodically the data sector and the copy sector must be erased. As seen in Section 4.2.3 this can add 90 ms or more. This delay can be minimized by using interrupts instead of polling to complete the EEPROM programming and to send the data out the UART.

To use EEPROM interrupts in the FEE driver define the symbol EEPROM_INTERRPTS. You can add this definition in the file feeConfig.h, or add it to Code Composer Studio's predefined symbols section. This causes fee.c to enable the EEPROM interrupt after each attempt to write to the EEPROM and disable the interrupt in FeeMainFunction(). The EEPROM module will generate interrupts on completion of either a read or a write to the EEPROM. The interrupt on the read is not necessary because the read value is returned immediately. By enabling and disabling the EEPROM interrupt the FEE driver avoids the unnecessary interrupts on EEPROM reads.

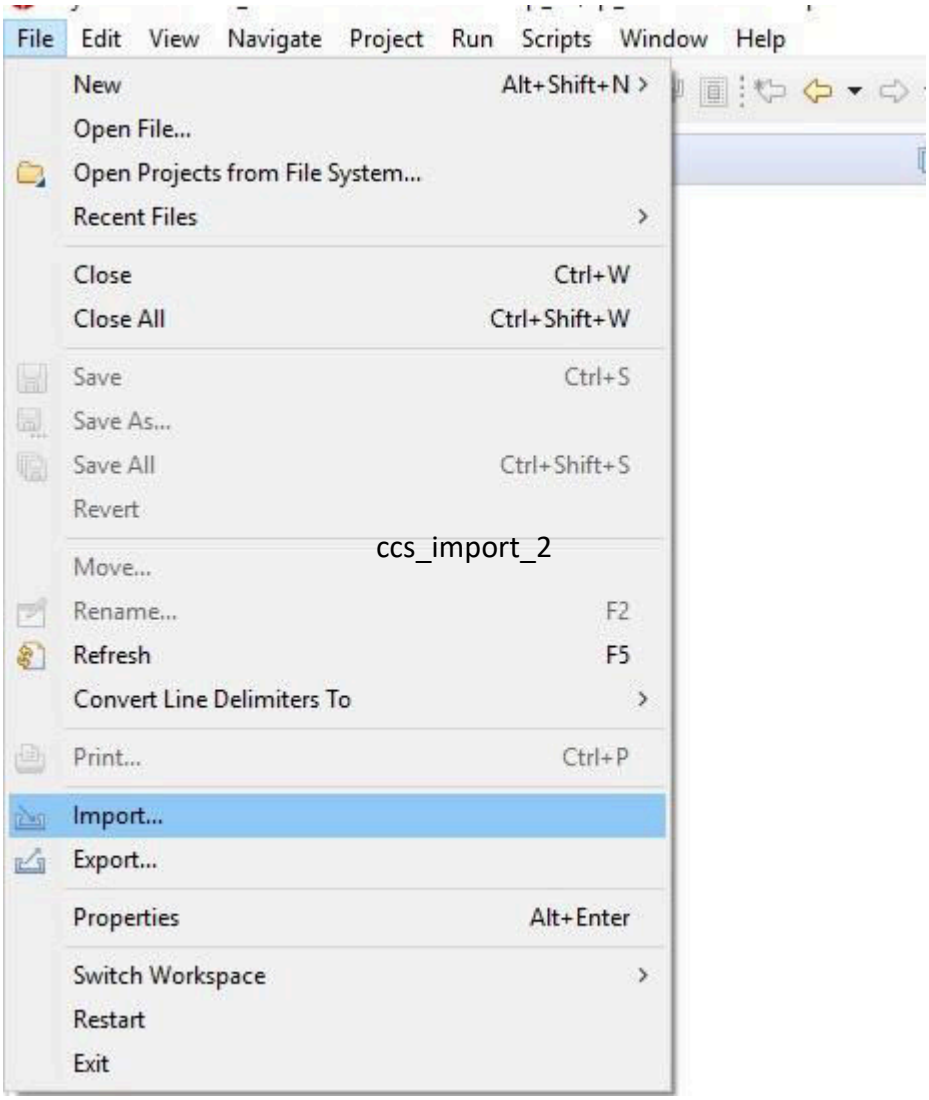
In the example "FeeExampleInt", the file main.c also includes a flash interrupt service routine, FlashIntRoutine(). This routine is statically added to the vector table by adding it to the statup_ccs.c file. The flash interrupt service routine checks that the source of the flash interrupt is from the EEPROM module and then calls FeeMainFunction().

In this example UART0 is also serviced by interrupts. This is done by defining the symbol "UART_BUFFERED" in Code Composer Studio's predefined symbols section and statically adding the function UARTStdioIntHandler() to startup_ccs.c.

5.3 Import, Program and Run Examples

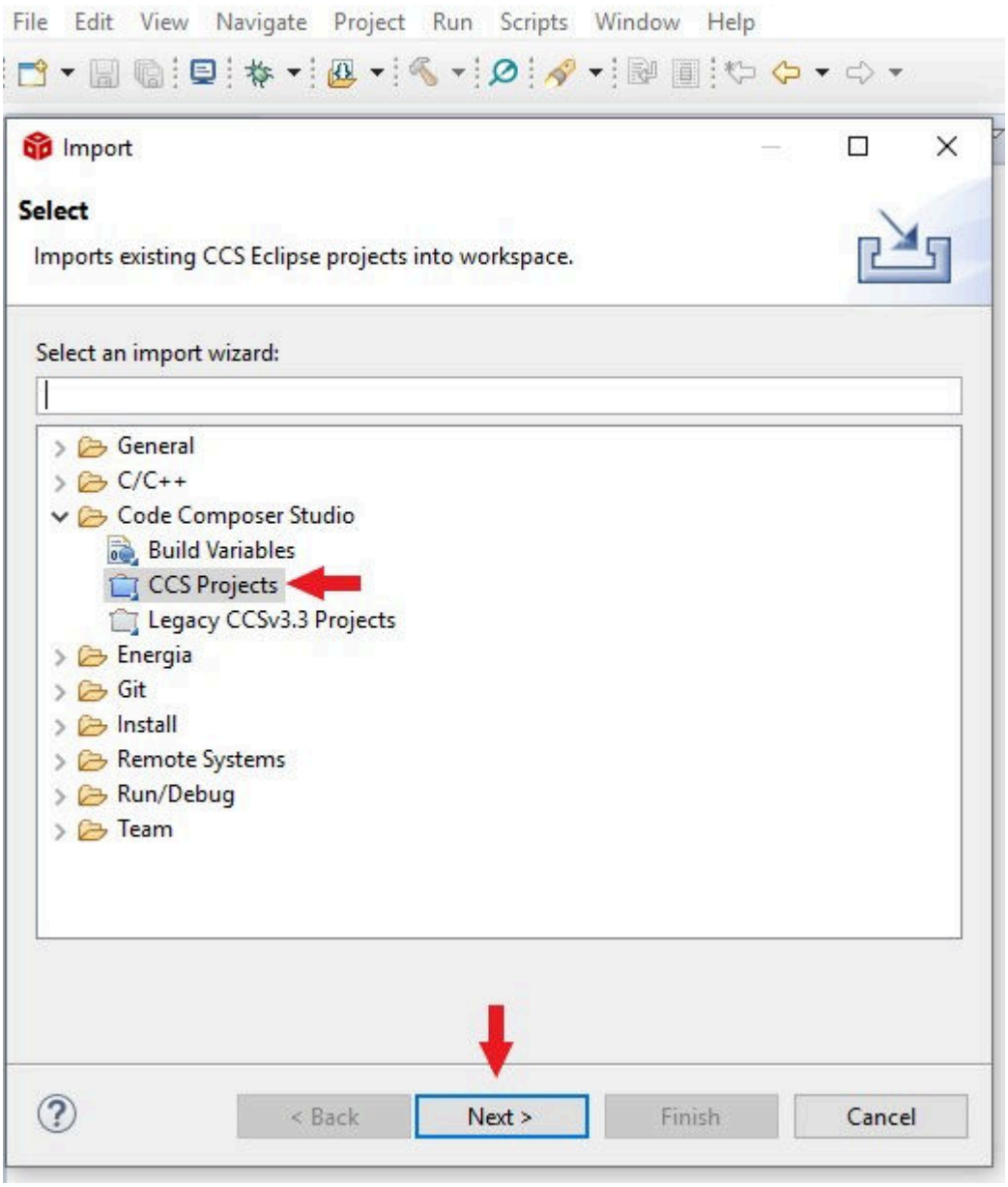
The examples can be downloaded from the following URL: www.ti.com/lit/zip/spma078. You can unzip the project or keep it in zip format. Both formats can be imported into CCS.

5.3.1 To Import the Project into CCS, First Select "File" → "Import"

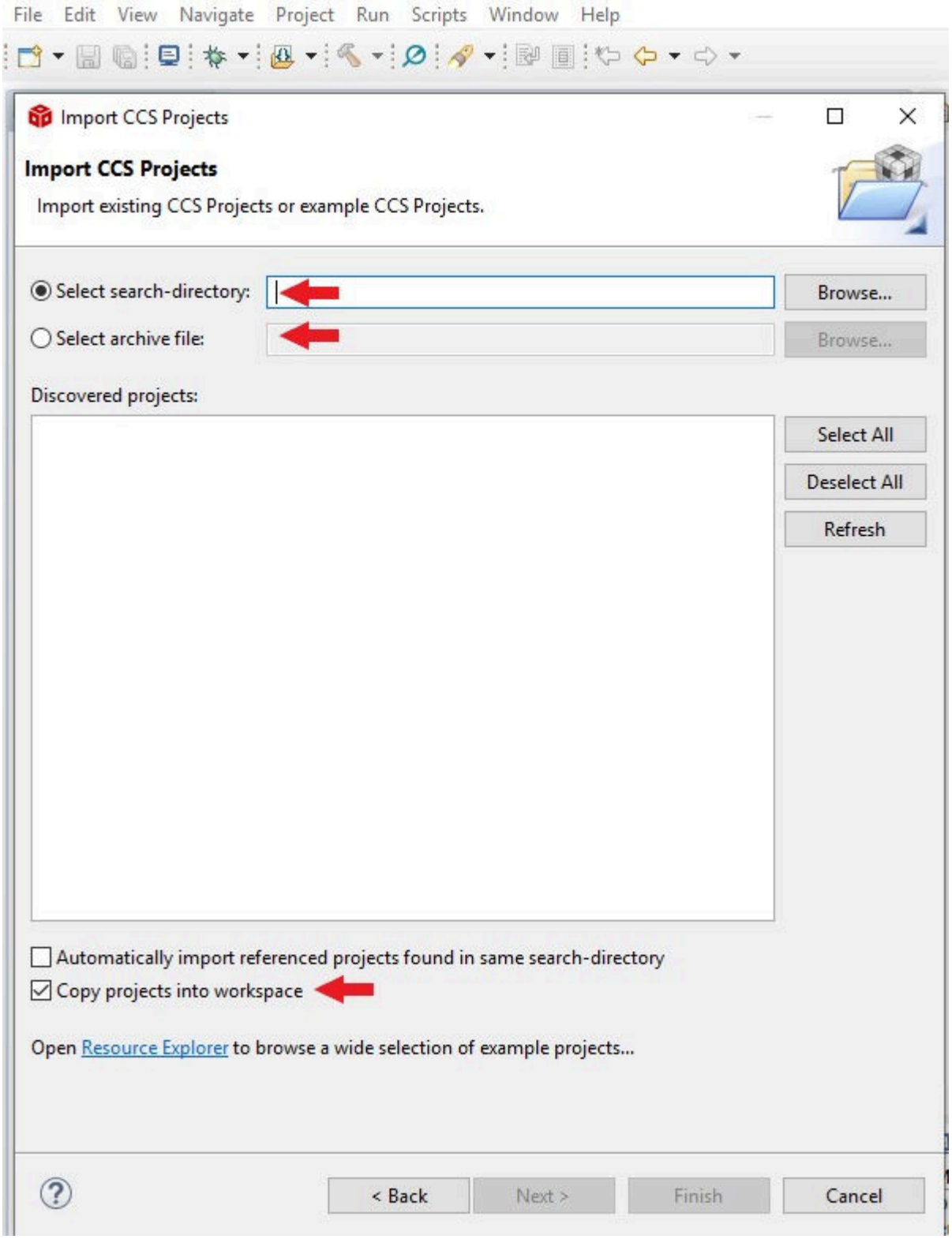


ccs_import_2

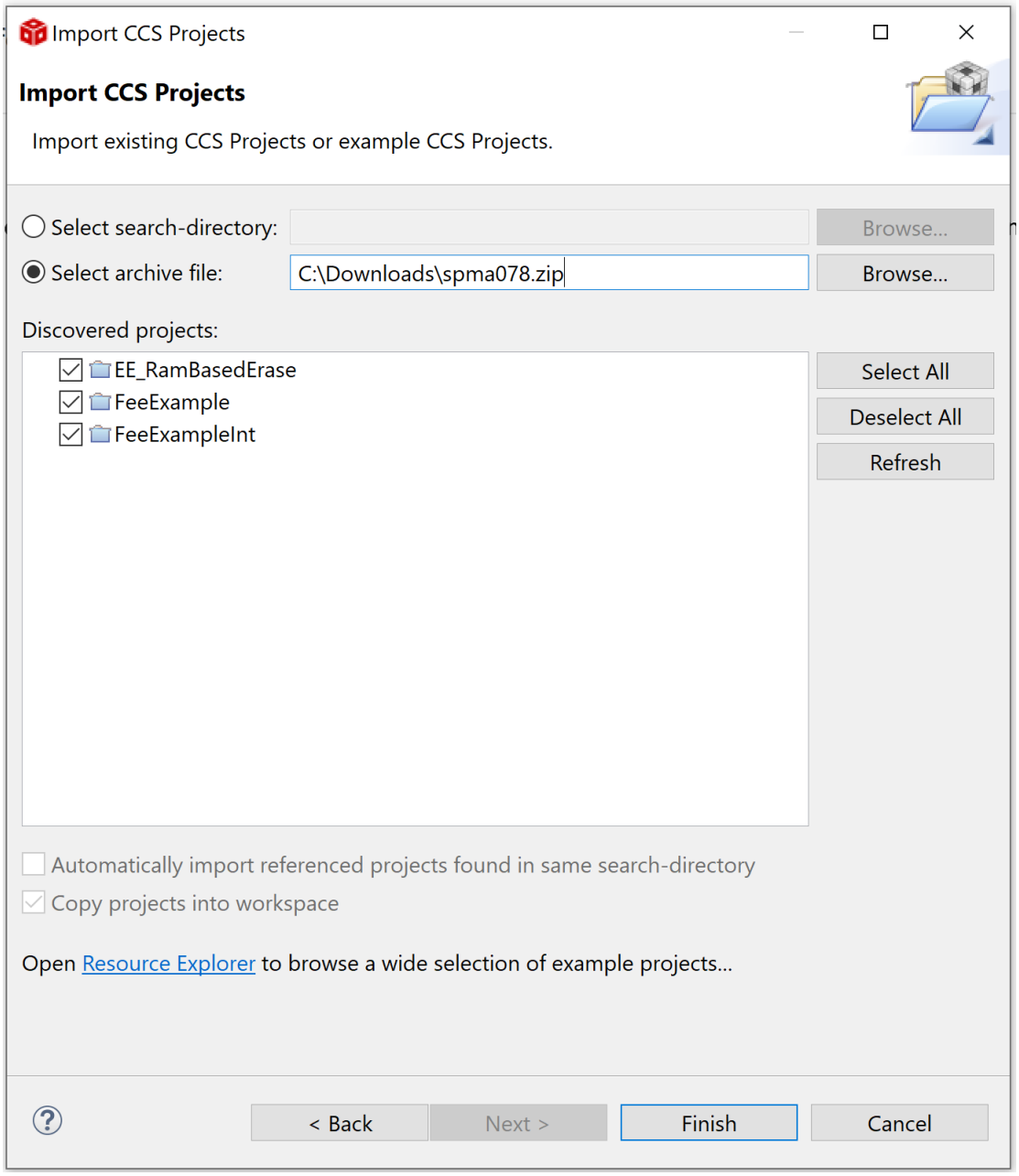
5.3.2 Select "CCS Projects" to Import the Example, Then Click "Next"



5.3.3 Provide the Path to Either the Unzipped Project by Selecting the First Radio Button or Import the Sip File Directly by Selecting the Second Radio Button. Click the "Copy Projects Into Workspace".

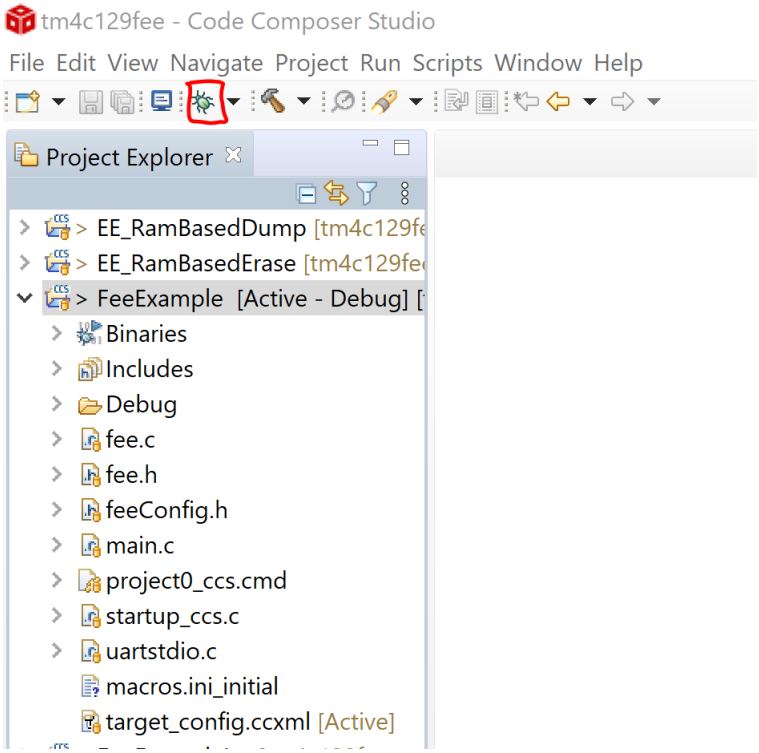


5.3.4 After the Project Path is Provided, it Will Show up as a Discovered Project. Click the "Finish" Button to Complete the Import.



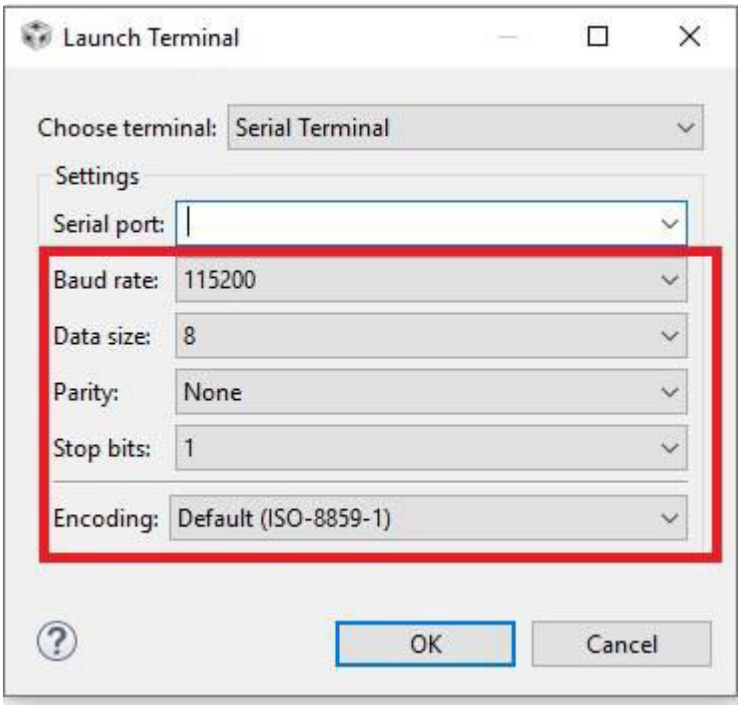
5.3.5 Program the Example into the Flash Memory and Start Executing the Example

With the LaunchPad connected to the PC via the USB cable, build the project and load the program by clicking the Debug icon.If you are new to CCS, see the Code Composer Studio User's Guide.



5.3.6 Configure the Serial Terminal Window

Once the example is loaded, the demo program is ready to run. The examples use the terminal window to display its output. Any serial terminal emulator may be used. CCS has a built-in terminal emulator that can be invoked with "View" -> "Terminal". The terminal emulator should be configured for 115200 Baud and 8-N-1.



5.3.7 Run the Example

When the example runs, LEDs D1 and D2 will be rapidly alternating on. The serial terminal will display the count status of each counter. Pressing SW1 or SW2 will increment the count and generate an output on the terminal.

```
Fee Example Version 1.00
Fee Version 1.00
 Data from dataset 0 is good
   Count = 0
   Cycle count = 32
 Data from dataset 1 is good
   Count = 0
   Cycle count = 32
Counter in dataset 0 set to 1
Counter in dataset 1 set to 1
```

6 Summary

The TM4C129x devices contain a Flash EEPROM Emulation module. Both the write-erase endurance and the integrity of congruent data in the case of power loss or reset can be increased by using this software FEE driver with the hardware module.

IMPORTANT NOTICE AND DISCLAIMER

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

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

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

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

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

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