TI-RSLK MAX

Texas Instruments Robotics System Learning Kit

Texas Instruments

Module 4

Lab 4: Software Design Using MSP432

Lab 4: Software Design Using MSP432

4.0 Objectives

The purpose of this lab is to interface a line sensor that the robot will use to explore its world.

1. You will learn logic, conditionals, and debugging in C.
2. You will write functions with input and output parameters.
3. You will implement logic and arithmetic functions.
4. You will implement consistency checks to make sure the data is realistic.
5. You will use an automated test approach called black-box functional testing to verify your algorithm is operating properly.

Good to Know: Implementing algorithms in software is an important task of all embedded systems. The manner in which you define, implement, and test the algorithm in this lab could be used to address many robotic control problems.

4.1 Getting Started

4.1.1 Software Starter Projects

Look at these three projects:
SineFunction (a simple implementation of sine),
ProfileSqrt (simple implementation of sqrt), and
Lab04_SoftwareDesign (starter project for this lab)

4.1.2 Student Resources

GP2Y0A21YK0F_IR_Distance_Sensor.pdf, datasheet for sensor

4.1.3 Reading Materials

Chapter 4, “Embedded Systems: Introduction to Robotics"

4.1.4 Components needed for this lab

All the components needed in the lab are included in the TI-RSLK Max Kit (TIRSLK-EVM kit), for this lab you will need only the MSP432-LaunchPad.

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R

4.1.5 Lab equipment needed (none)

4.2 System Design Requirements

Throughout the course you will acquire knowledge that will allow you to solve many robot challenges. The goal of this lab is to build some of the software components that the robot system will need to explore a world that has walls, as shown in Figure 1. In this lab, we will learn how to build C functions to gather information that will allow the robot to navigate and reach the treasure or goal. For the actual challenge you will consider a robot with three distance sensors, and use the distance sensors to collect information on location and make necessary decisions based on the scenarios.

Good engineers employ well-defined design processes when developing complex systems. When we work within a structured framework, it is easier to prove our system works (verification) and to modify our system in the future (maintenance). As our software systems become more complex, it becomes increasingly important to employ well-defined software design processes. This course focuses on real-time embedded systems written in C. At first, it may seem radical to force such a rigid structure to software. We might wonder if creativity will be sacrificed in the process. True creativity is more about effective solutions to important problems and not about being sloppy and inconsistent. Because software maintenance is a critical task, the time spent organizing, documenting, and testing during the initial development stages will reap huge dividends throughout the life of the software project. Consider these steps

- *Test it now*: When we find a bug, fix it immediately.
- *Plan for test*: Consider debugging at all stages of design.
- *Get help*: Use the available tools for design and debug.
- *Divide and conquer:* Use creativity to break it into simple pieces

Lab 4: Software Design Using MSP432

The most important skill you should develop in this lab is mechanisms to facilitate software testing.

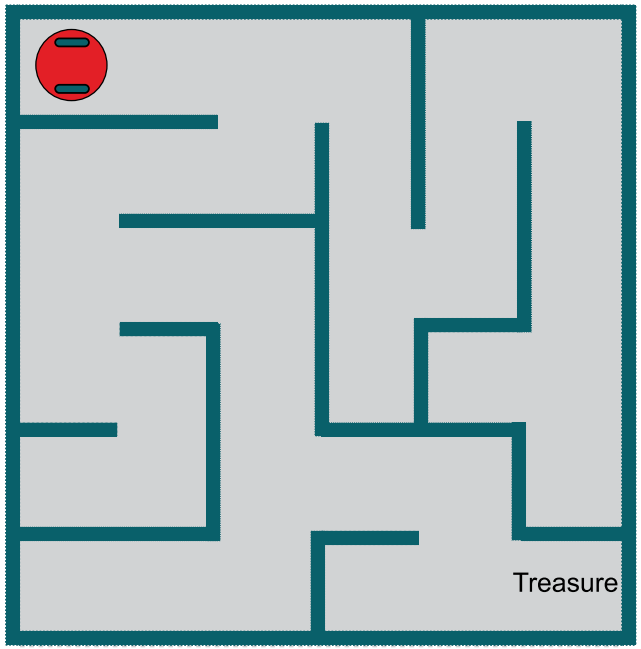

Figure 1. Possible robot challenge of exploring the world.

In Module 15, we will interface the actual distance sensors to the analog to digital converter (ADC) on the MSP432. The ADC converts analog voltages (0 to 3.3V) into digital values (0 to 16383). The first task in this lab is to develop a function in C that **converts** raw ADC samples generated within the TI LaunchPad development board.

Note: In Module 15 you will use a distance measuring sensor unit composed of an integrated position sensitive detector and an IR sensor. This is also called a Proximity sensor, three of which will be placed on the robot to measure distances to the wall.

Let **n** be a 14-bit sample from the ADC, and **D** be the distance in mm. The basic form of this nonlinear transfer relation is

$$D = 1195172/(n - 1058)$$

where 1195172 and -1058 are calibration coefficients to be empirically determined in the ADC lab (Module 15). The prototype for your function is

int32_t Convert(int32_t n);

The second task (software algorithm) needed by the robot is to use three distance numbers to determine and classify the situation into one of many possible scenarios. Let us assume that the robot has three distance sensors: **left**, **center**, and **right**, and each sensor will measure the distance from the center of the robot to the wall in mm. There will be a single reference point on the robot, and the three distances will be measured from that common reference, as shown in Figures 2 and 3. These two figures show eight possible scenarios as the robot approaches a decision point.

Software is layered with I/O at the lowest layer. This **Convert** function will reside in this lowest level. This software module will abstract the details, separating what is does (measure distance) from how it works (nonlinear, ADC-based, IR distance sensor).

In a higher-level module, the software will decide to go straight, turn left, turn right, or turn around. We will also worry about being too close to the wall. In this lab, you do not take distance measurements from an actual sensor. Rather, you will take three distance numbers (left, center, right) and determine which of the possible scenarios is most likely.

Lab 4: Software Design Using MSP432

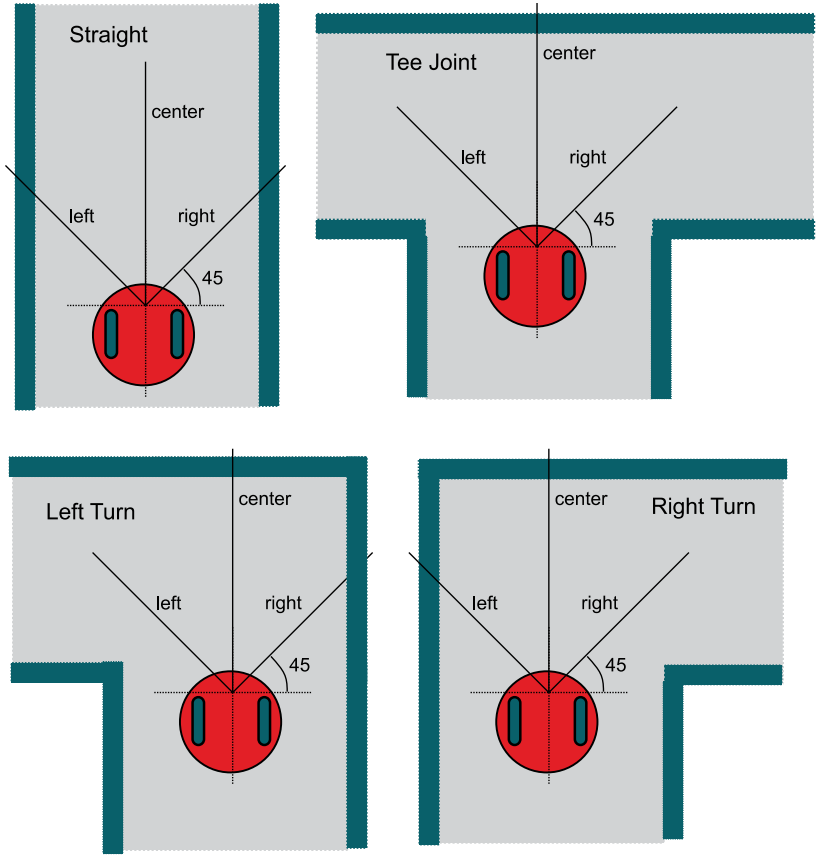

Figure 2. Four possible scenerios as the robot approaches a decision point.The three variables (left, center, and right) are defined as the distance from the center of the robot to the wall.

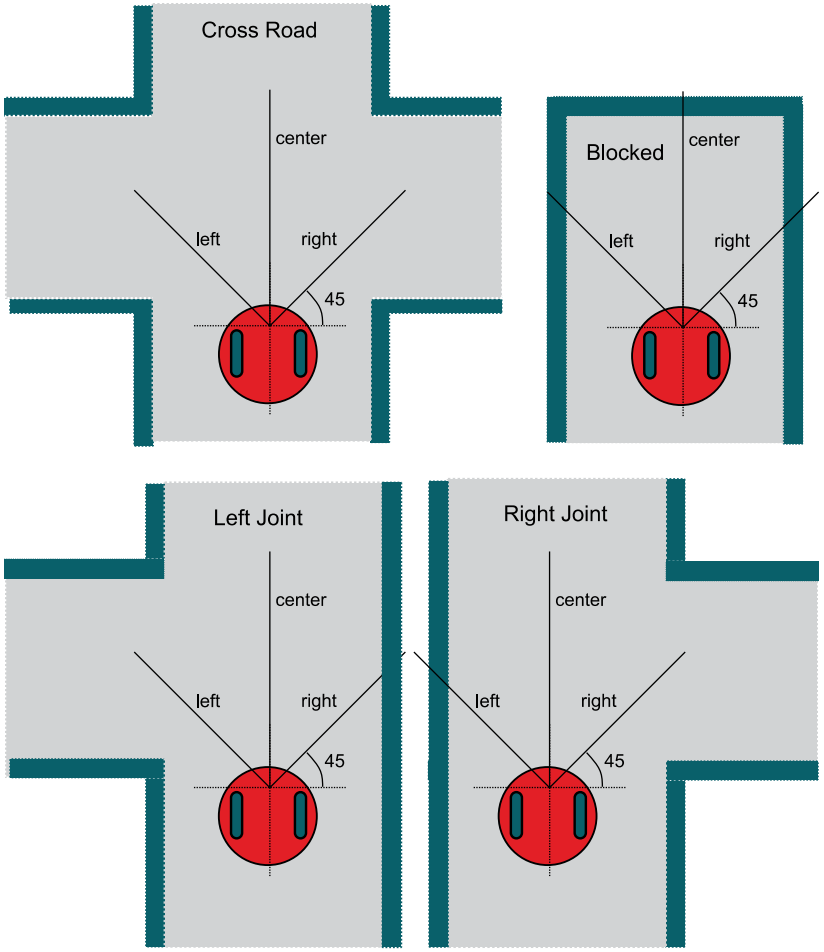

Figure 3. Four more scenerios as the robot approaches a decision point.

We begin to define the algorithm design with the most important classification, the danger conditions.

The algorithm will return a **LeftTooClose** (4) error if the left sensor is less than 212, and return a **RightTooClose** (2) error if the right sensor is less than 212. The algorithm will return a **CenterTooClose** (1) error if the front sensor is less than 150 mm. It will be possible for there to be multiple simultaneous

danger conditions. For example, 5 will signify both too close to center and too close to left. 7 will mean all three directions are too close.

First we will consider the right and left sensors. In this lab, we will use **#define** statements to specify the distance thresholds. Once the robot and arena are built, these numbers will need to be tuned. In this example we use numbers derived from a road that is 400 mm wide and side sensors that are placed at 45 degrees.

If the robot were in the middle of a road with the straight or blocked scenarios, both side sensors (placed at 45 degrees) would read 283 mm. If the robot were within ±50mm from the center of the road with the straight or blocked scenarios, the side sensors could range from 212 to 354 mm. The 354 threshold will be used to classify whether or not it is possible to turn left or right at the next intersection. Less than 354 means turn path not possible; 354 or above means turn path is possible.

Finally, consider the center sensor. As the robot approaches the intersection, the center sensor will be used to classify the difference between {Blocked, Right Turn, Left Turn, and Tee Joint} (center sensor less than 600 mm) and {Straight, Right Joint, Left Joint, and Cross Road} (center sensor more than 600 mm). Because there could be a long straight road, there is no maximum acceptable value for the sensors.

Note: The particular sensor has a measurement range from 10 to 800 mm. However, for this algorithm the smallest distance will be 50 mm because the distance is specified to the center of the robot, not from the sensor.

You are asked to develop an algorithm that will enable your robot to explore the arena (maze) and provide the necessary classification. Assume you will take three distance measurements with the sensors placed on the robot as inputs and return the most likely scenario based on the above criteria.

There are 16 possible outputs of the classification algorithm. To make the software more readable, we define an **enumerated data type** for the return parameter. In this case, we assign specific integers for each possibility. This allows us to combine 1, 2, and 4 to represent the 7 possible danger situations. For example, a 5 means left sensor too close AND to center sensor too close. In addition you should return an **Error** if any input is below 50 or greater than 800. In particular, we define

```
enum scenario {
  Error = 0,
  LeftTooClose = 1,
  RightTooClose = 2,
  CenterTooClose = 4,
  Straight = 8,
  LeftTurn = 9,
  RightTurn = 10,
  TeeJoint = 11,
  LeftJoint = 12,
  RightJoint = 13,
  CrossRoad = 14,
  Blocked = 15
};
typedef enum scenario scenario_t;
```

We will use **#define** statements to specify the bounds to make it easier to understand the classification algorithm.

#define SIDEMAX 354 // largest side distance to wall in mm
#define SIDEMIN 212 // smallest side distance to wall in mm
#define CENTEROPEN 600 // distance to wall between open/blocked
#define CENTERMIN 150 // min distance to wall in the front

The prototype for your classification algorithm is

scenario_t Classify(int32_t Left, int32_t Center, int32_t Right);

4.3 Experiment set-up

This lab uses the LaunchPad without any input/output.

4.4 System Development Plan

4.4.1 Functions and debugging

Build and debug the **SineFunction** example. Using the debugger, observe the input and output parameters of the function while you single step through the main program. Run the program and observe the results in the array. Explain the purpose of the two while loops at the beginning of **fsin**. Explain the purpose of the if-then-else statements in **fsin**. Prove that the **fsin** function operates properly.

Lab 4: Software Design Using MSP432

Build and debug the **ProfileSqrt** project. Using the debugger, place a breakpoint inside the loop of the **sqrt** function and observe the values of **n**, **s**, and **t** each time **t** is updated for one execution of the **sqrt** function. Determine after how many iterations does the function converge. Suggest ways to make the program execute faster.

4.4.2 Distance conversion

Using the TI's LaunchPad development board, write a C function that converts raw 14-bit ADC data to distance in mm. Please note each GP2Y0A21YK0F sensor and each MSP432 will be slightly different, in the program we will use **#define** statements to encapsulate the calibration parameters.

Note: The actual distance sensors GP2Y0A21YK0F will be interfaced and calibrated as part of Lab 15.

#define IRSlope 1195172
#define IROffset -1058
#define IRMax 2552

The maximum measurement distance for the sensor is 800 mm, so if the ADC value is less than 2552 (IRMAX), your function should return 800. You can use **Program4_1** to test your **Convert** function. You will find Program 4_1 in the starter project for this lab. This approach is called **functional testing**. This test program contains 16 test cases (inputs and expected outputs). The expected results are plotted as Figure 4.

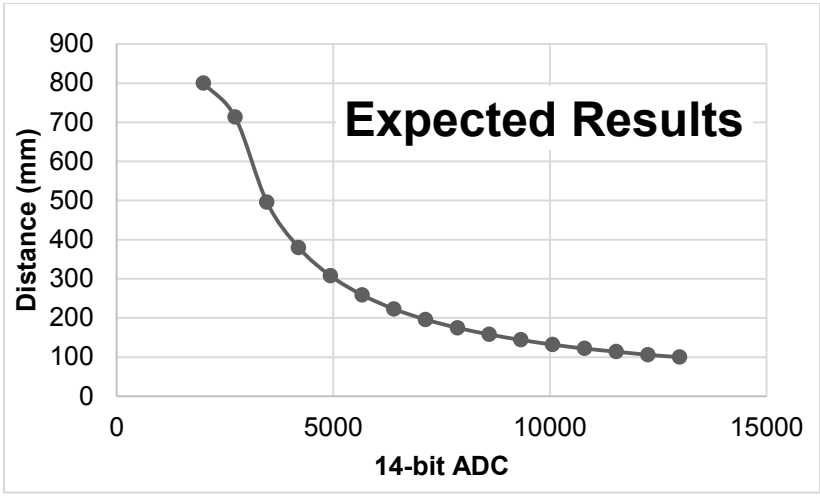

Figure 4. Expected results for the GP2Y0A21YK0F conversion function.

```
// Program 4_1 used to test the Convert function
int32_t const ADCBuffer[16]={2000,2733,3466,4199,4932,
 5665, 6398, 7131, 7864, 8597, 9330, 10063, 10796,
 11529, 12262, 12995};
int32_t const DistanceBuffer[16]={800,713,496,380,
 308,259,223,196,175,158,144,132,122,114,106,100};
void Program4_1(void){int i;
int32_t adc,distance,errors,diff;
  errors = 0;
  for(i=0; i<16; i++){
    adc = ADCBuffer[i];
    distance = Convert(adc); // call to your function
    diff = distance-DistanceBuffer[i];
    if((diff<-1)||(diff>1)){
      errors++;
    }
  }
  while(1){};
}
```

To run this test program, rename **Program4_1** to **main**, and rename the actual **main** to **main2**. Run **Program4_1** and compare your results with expected values. It is ok if your results differ by ±1 (which could be due to rounding).

4.4.3 Classification algorithm

The first step in solving a complicated problem is to break it into pieces. Begin by creating eleven different classification algorithms, one for each of the 15 scenarios. Use flowcharts or pseudo code to define each algorithm. Example, you could define

CenterTooClose if (Center < CENTERMIN)

or

Blocked if (SIDEMIN ≤ Left < SIDEMAX)
and (SIDEMIN ≤ Right < SIDEMAX)
and (CENTERMIN ≤ Center < CENTEROPEN)

A good flow for the example described in section 4.2 is to first work out the **Error** conditions. Next, consider the danger conditions, and return 1 – 7 if any combination of danger conditions exist.

Next, consider remaining possible values for the three distance inputs. If there are any possible input value combinations that match none of the eight scenarios shown in Figures 2 and 3, then expand the selection criteria to satisfy "the most likely" possibility. If you have input patterns that result in multiple selections for the same input data, reduce the selection criteria to remove the overlap, again satisfying "the most likely" possibility.

For the **convert** function we used a set of 16 test cases, with input values that were linearly separated from 2000 to 12995 together with expected output values. For the **Classify** function, each of the three possible inputs can vary from 50 to 800. Therefore, there are 751^3 (423,564,751) possible inputs. An exhaustive test would evaluate them all. However, due to the nature of the problem, we can reduce the input values to a small subset of values around the threshold values. Using knowledge of how the system works to select strategic values to test is called **corner cases**. In particular, we can reduce the number of test values from 751 down to 18 with minimal loss of testing accuracy. In particular, we will only test values that are ±1 from the threshold values of 50, 150, 212, 354, 600, and 800.

int32_t const CornerCases[18]={49,50,51,149,150,151,211,212,213,353, 354,355,599,600,601,799,800,801};

Using corner cases reduces the search space from 751^3 to 18^3 (5832).
The second approach to testing used for this function is the availability of a working solution. Your instructors have written a solution to the classify algorithm and hidden its implementation in object form (as ***Solution.obj***). You can however call the instructor's function to see what the correct classification should have been for any possible input. The prototype for this solution is

scenario_t Solution(int32_t Left, int32_t Center, int32_t Right);

You can use this **Program4_2** to test your **Classify** function. This program tests all 5832 corner cases. The expected result is determined by calling the instructor's ***Solution***.

```
// Program 4_2 tests the corner cases
int32_t errors;
void Program4_2(void){
  scenario_t result,truth;
  int i,j,k;
  int32_t left, right, center; // sensor readings
  errors = 0;
  for(i=0; i<18; i++){
    left = CornerCases[i];
    for(j=0; j<18; j++){
      center = CornerCases[j];
      for(k=0; k<18; k++){
        right = CornerCases[k];
        result = Classify(left,center,right); // yours
        truth = Solution(left,center,right);  // correct
        if(result != truth){
          errors++;
        }
      }
    }
  }
  while(1){
  }
}
```

4.5 Troubleshooting

Convert doesn't work:

- Using **Program 4_1**, find an input pattern that does not work, write a main program that calls your function with just that input, and single step your program comparing your internal calculations with expected values.
- If you are still having bugs, we suggest you break the calculation into multiple steps (one arithmetic operation per line of C), this way you can single step across each calculation.

Classify doesn't work:

- Using Program 4_2, find an input pattern that does not work, compare your output with the expected output. Using Figures 2 and 3, reconsider which scenario should have matched that input pattern. Write a main program that calls your function with just that input, and single step your solution to find the difference between your function and expected results.
- If you are still having bugs, consult with your instructor and/or fellow students. You may be interpreting the problem in a different way as the instructor solution.

4.6 Things to think about

In this section, we list thought questions to consider after completing this lab. These questions are meant to test your understanding of the concepts in this lab.

- How does the software handle the nonlinear response of the distance sensor, as shown in Figure 3?
- We used signed numbers even though all the distances were unsigned. If you tried implementing convert with unsigned parameters you will get a compiler warning (and it still would have worked). Why does the compiler object to unsigned for this function?
- It is often the case that testing software is actually a more difficult job than writing the software in the first place. List the testing procedures introduced in this lab.
- Why did we allow for ±1 difference on the **Convert** function?
- What kind of crazy situation could the robot be in to cause a classification of 7?

4.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. You could extend the system or propose something completely different. For example,

- Consider exhaustive testing as shown in Program 4_3 (i.e., test all possible input values). This problem may take over 16 hours to complete. What are the advantages of exhaustive testing?
- Redesign the classification system using only two distance sensors and a front bumper switch.
- Redesign the classification system using four sensors.
- With five distance sensors you could also calculate angle to the left and right walls.
- Consider how you could test the **Classify** function if there were no solution available. For example, what could you do for this lab if you were to combine all the solutions to the lab from the entire class without "looking" at each other's solution?

4.8 Which modules are next?

We will use the next few labs to create additional components we will need to control the robot. The input/output are an important component of an embedded system. The following modules will build on this module:

Module 5) Begin construction of the robot, including battery and voltage regulation
Module 6) Learn how to input and output on the pins of the microcontroller
Module 7) Study finite state machines as a method to control the robot
Module 8) Interface actual switches and LEDs to the microcontroller. This will allow for more inputs and outputs increasing the complexity of the system.
Module 9) Develop a simple PWM output to adjust duty cycles

Lab 4: Software Design Using MSP432

4.9 Things you should have learned

In this section, we review the important concepts you should have learned in this module:

- Use functions to provide software abstraction
- Perform logic functions using AND and OR
- Perform arithmetic calculations with addition, subtraction, multiplication, and division
- Use #define to improve readability of the software
- Use enum and typedef to create new data types
- Make decisions with if-then statements
- How to handle error conditions
- Use the debugger to single step and visualize variables
- Perform functional testing
- Use corner cases to reduce the testing time

ti.com/rslk

IMPORTANT NOTICE AND DISCLAIMER

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

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, 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 (www.ti.com/legal/termsofsale.html) 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.

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