TI-RSLK

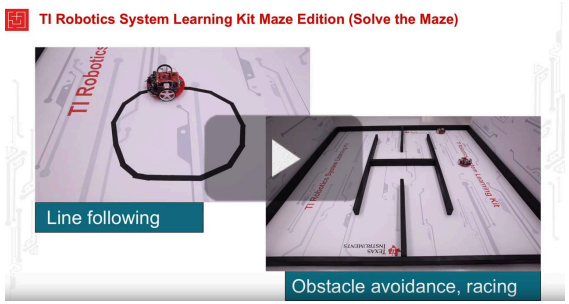
Texas Instruments Robotics System Learning Kit
The Maze Edition - Basic

Texas Instruments

Table of Contents

Preface

Texas Instruments Robotics System Learning Kit
The Maze Edition



Preface

The ultimate goal of the learning kit is to design, build, and test a robot system capable of solving complex tasks. One possible robot is shown in Figure 1. Example challenges include exploring a maze, racing autonomously, finding an object, and following a line. However, it is not the final robot that matters, but the educational journey that discovers a wide range of engineering principles along the way. Rather than just providing the robot kit and a challenge to solve, this curriculum follows an educational road map that intentionally exposes deep learning along the way.

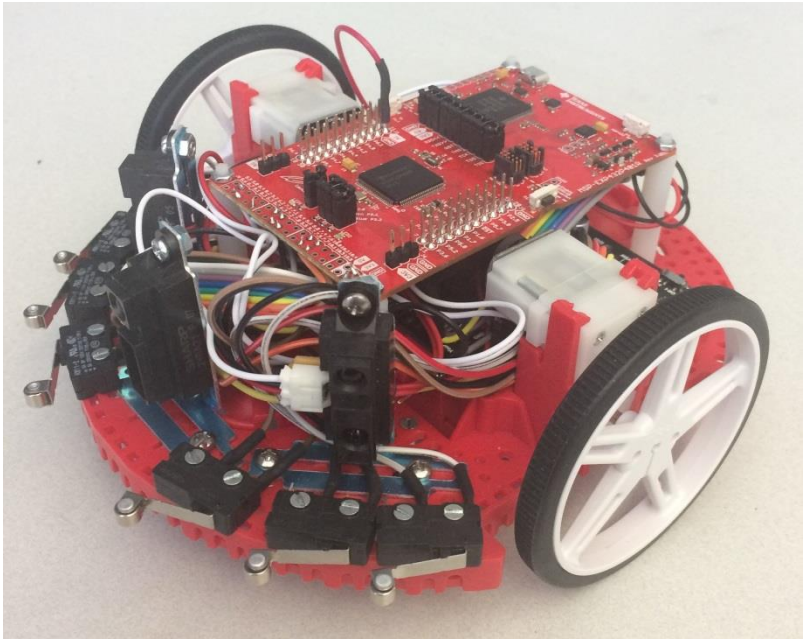


Figure 1: TI-RSLK Maze Robot

The **EE skills** you will learn include voltage, current, power, energy, batteries, resistors, capacitors, transistors, diodes, and DC motors. You will learn how to use a voltmeter, an ohmmeter, a current meter, and an oscilloscope.

This is an **embedded systems** curriculum; hence you will interface numerous devices to the MSP432 microcontroller. In particular, you will interface LEDs, switches, a line sensor, DC motors, tachometers, and an LCD. Your microcontroller hardware/software skills will include pulse-width modulation (PWM), flash read-only memory, periodic timers, edge-triggered interrupts, digital to analog converter (DAC), analog to digital converter (ADC), synchronous serial communication, and asynchronous serial communication.

A significant component of the curriculum involves software development. You will develop **software skills** in multithreading, data structures, debugging, linked lists, semaphores, and first in first out (FIFO) queues. You will learn how to use a logic analyzer for interface testing, and thread profiling.

This is a lab-based curriculum. However, there are numerous **fundamental concepts** to master, such as the Nyquist Theorem, the Central Limit Theorem, digital filtering, and Little's Theorem.

The overriding theme of this curriculum is to teach **systems design** in a bottom up fashion. We begin with simple components so that you learn fundamentals. A system is created by combining multiple components with the appropriate hardware and software interfaces. Once you master the fundamentals of one component, its operation can be abstracted into a set of high-level functions. Separating how a component works (low-level implementation) from what it does (high-level abstraction) is the key for developing complex systems. Obviously, the most important system in this curriculum will be the robot. However, there will be other systems like a security system, a traffic light control system using a finite state machine, Bluetooth Low Energy (BLE) communication system, and the Wi-Fi-based internet of things (IoT) system.

A system is comprised of subsystems connected together to solve a unified objective. An effective approach to teaching systems is to begin with very simple components. First, one completely understands how the component works. Second, one creates an abstraction that separates what is does from how it works. Third, components are interfaced together to create a new more complex system.

Preface

The terms system, subsystem, and component are used here interchangeably.

As you can see from Figure 2, there are twenty modules in the curriculum. Each module is relatively independent, and you can thread together modules to create a particular learning experience for your students.

Each module has:

- Introduction to module (1 page)
  - Overview
  - Educational objectives
  - Prerequisites, bullet list linking to other modules
  - References
- Class lecture PowerPoint slides (one to three files)
- Screen capture video with audio of PowerPoint (one to three videos)
- Class activity, homework exercises or practice problems.
- Lab document
- Hardware needed,
  - BOM excel file of parts
  - Circuit diagrams in CircuitMaker.

- Lab solution for faculty to access, not available to students.
- One to three videos of finished lab
- Quizzes
- Quiz solutions for faculty to access, not available to students.

The most important document is the lab manual. Performing labs results in the design, construction, and testing of the robot system. To find the circuit diagrams, create an account in Circuit maker. Launch the application, under projects select Tags, and search MSP432. You will find starter circuits for each lab that has hardware.

The robot challenge document lists some possible final projects for the course. Most users of this curriculum will pick and choose a subset of the modules, allowing the user to focus on which concepts they wish to learn (or teach). Challenges are sorted by the set of sensors and actuators that are required.

Robot Features (Full set, Advanced Kit) :

- Robot Chassis with 2 DC motors and wheels
- 6 AA NiMH batteries
- Motor driver and power distribution board (MDPD) with motor drivers and voltage regulator to power your system
- 3 IR distance sensors
- 6 touch/bump sensors
- 8 line sensors
- 2 tachometers
- Tachometer
- BLE or Wi-Fi

Course prerequisites:

- Algebra and college physics
- Basic knowledge of computers and architecture
- C programming

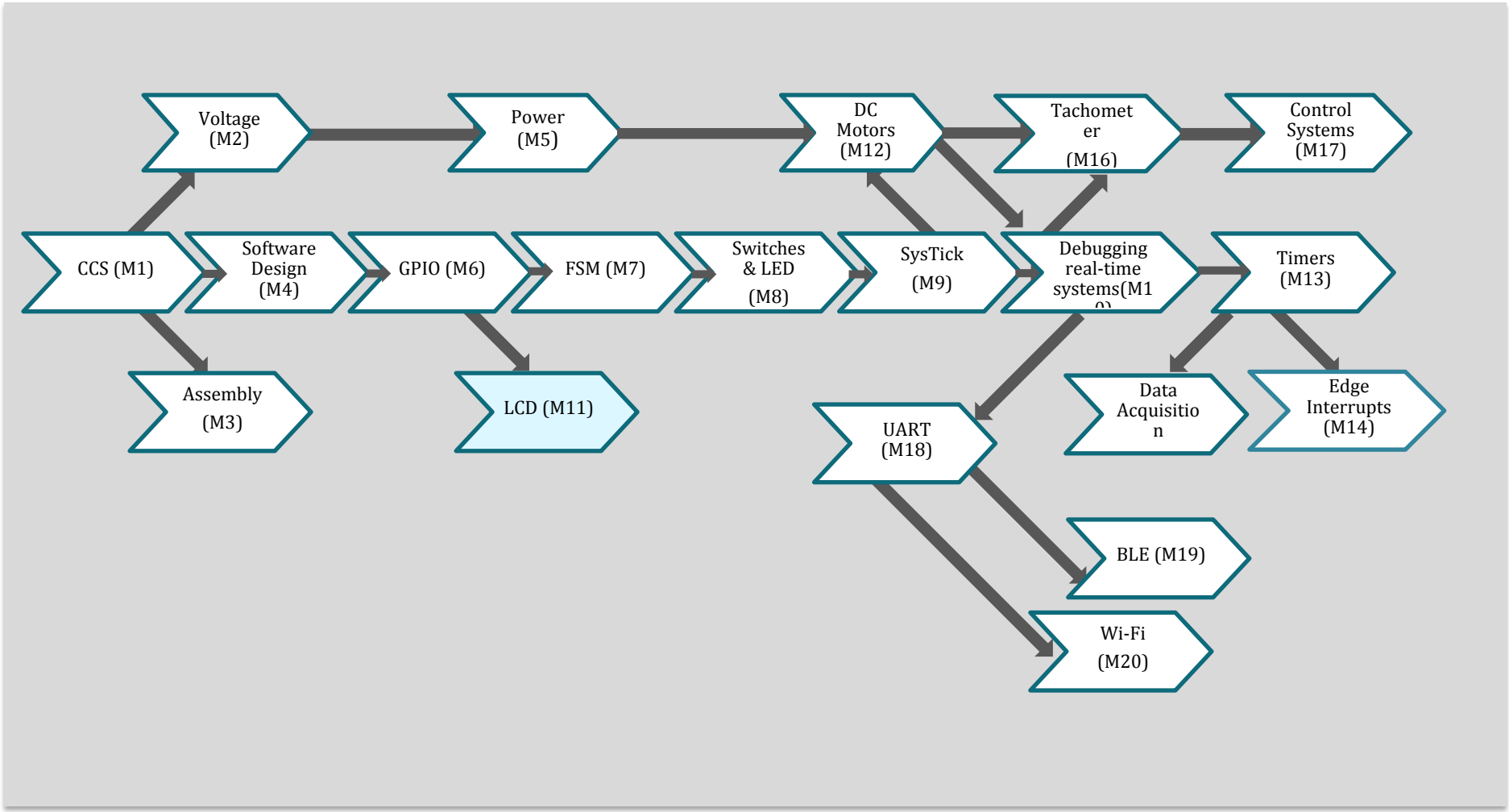


Figure 2. TI-RSLK: Learning Modules & Curriculum Pathways

Modules:

<CCS> 1. Running Code on the LaunchPad using CCS
Prerequisites: none
Equipment: LaunchPad
Theory: how to install and configure CCS for this class
Lab: installing CCS, MSP432 drivers, and running the TExaSdisplay logic analyzer

<Voltage_Current> 2. Voltage, Current and Power
Prerequisites: none
Motivation: The hardware interfaces for the robot involve voltage, current and power
Tools: Voltmeter, current meter, ohmmeter, signal generator, oscilloscope
Equipment: 220 470 22k 33k ohm resistors, 0.47uF 10uF capacitors, voltage supply, 10-mA LED and 2-mA LED
Theory: Resistor, Ohm's Law, LED voltage current relationship
Lab: Characterize LED, Measure Reactance

<Assembly> 3. ARM Cortex M
Prerequisites: <CCS>
Motivation: teaching assembly will help understand how it works, and how to debug
Tools: CCS
Equipment: LaunchPad (20-instruction subset of Cortex M)
Theory: machine code, registers, execution, bus, memory, simple I/O
Lab: assembly language programming
Installing and running assembly language

<SoftwareDesign> 4. Software Design using MSP432
Prerequisites: <CCS>
Motivation: most of the labs are C programmed on the MSP432, C programming is a prerequisite to the class, but an introduction to C code on the MSP432 is appropriate
Tools: CCS
Equipment: LaunchPad using built-in switches and LEDs
Theory: Using typical input parameters for the robot, perform, logical operations of AND, OR, EOR, shift, add, subtract, multiply, divide, variables, and functions
Lab: Simple C programming converting ADC-inputs to calibrated distance. Given three distance measurements, implement a classification algorithm to interpret the robot world

<Power> 5. Battery and Voltage Regulation
Prerequisites: <Voltage_Current>
Motivation: Robot is battery powered; robot motor controller board has power regulation
Tools: Voltmeter, current meter, ohmmeter
Equipment: Two power resistors 5W 10-ohm resistor and 22-ohm 5W resistor, rechargeable battery (4.8V/10ohm) is 0.48A, 2A-hr battery lasts 4hr Robot with power regulation/motor driver board
Theory: Resistor, Ohm's Law
Theory: total energy in battery voltage current relationship while V>regulator minimum
Lab: battery power, calculations, measurements

<GPIO> 6. GPIO – MSP432
Prerequisites: <Logic>
Motivation: robot line sensor is needed for line following
Tools: CCS Voltmeter, oscilloscope,
Equipment: LaunchPad, integrated line sensor
Theory: conversion light to voltage, direction registers, input, output, friendly (This connects to the maze robot).
Lab: input from line sensor, output to build-in LED
Input line sensor, detect position relative to a black line on a white field

<FSM> 7. Finite State Machines
Prerequisites: <Logic><GPIO>
Motivation: FSMs are an effective solution to robotic functions
Tools: CCS
Equipment: LaunchPad using built-in switches and LEDs
Theory: loops, decisions
Lab: Very simple C programming, 2-input, 2 output FSM

<Switches_LED> 8. Interfacing Input and Output
Prerequisites: <GPIO>
Motivation: robot touch sensors will be switches, LEDs provide debugging outputs for the robot
Tools: Voltmeter, current meter, ohmmeter, CCS
Equipment: switches, LEDs, resistors, LaunchPad
Theory: GPIO, LED, positive/negative logic, pullup/pulldown, input/output
Lab: Input from switches, output to LED

<SysTick>9. SysTick Timer
Prerequisites: <Switches_LED>
Motivation: introduction to time, introduce the concept of PWM that will be needed to drive the DC motor later; this is also introduces the need for interrupts, because this one task will require 100% processor utilization
Tools: CCS, logic analyzer,
Equipment: LaunchPad, LED
Theory: Introduce the need for the microcontroller to manage time. Define processor utilization. Use SysTick to create time delays. Use time delays to create a PWM signal. Use PWM to control power delivered to an actuator (LED).
Lab: A GPIO port is connected to an LED and the software controls brightness of the LED using duty cycle. The software varies the duty cycle sinusoidally (table look up) to make the LED appear to be "breathing". Add a resistor and capacitor, and then observe the sinusoidal output on the oscilloscope.

<Debug>10. Debugging Real-time Systems
Prerequisites: <SysTick>
Motivation: system level design/debug; students need effective debugging skills. This will mimic the process students will use to design/develop/debug their robot. Eventually, students will place the robot in the maze and hit go. The robot runs for a while autonomously. After the run, students can reconnect the USB cable and upload parameters measured during the run. This module will introduce interrupts and use SysTick to perform the line-sensor measurements in the background.
Tools: CCS
Equipment: LaunchPad, line-sensor, bump sensors
Theory: RAM versus ROM, arrays, pointers, periodic interrupts
Lab: record bump and line sensor data into arrays in Flash

<LCD> 11. Liquid Crystal Display (optional)
Prerequisites: <GPIO>
Motivation: optional for robot, but makes the course accessible for other non-robotic applications; if used with the robot, the LCD can help with debugging during the stand-alone running.
Tools: Logic analyzer, CCS
Equipment: Nokia 5110, proto-board, LaunchPad
Theory: SPI interface, graphics
Lab: Display text and graphics

<Motors> 12. DC motors
Prerequisites: <Power><SysTick>
Tools: Voltmeter, current meter, oscilloscope, CCS
Equipment: LaunchPad, Motor driver board (MOSFET, resistors, diodes), DC motor
Theory: Brushed DC motor, PWM
Lab: Open loop DC motor output, measure speed versus duty cycle, extends the simple PWM built in <SysTick>

<Timers> 13. PWM and Periodic interrupts using Timers
Prerequisites: <Motors><Debug>
Motivation: periodic interrupts are a simple way to create PWM outputs to DC motors
Tools: Voltmeter, current-meter, oscilloscope, logic analyzer, CCS
Equipment: LaunchPad, DC motor on robot
Theory: timers, interrupts, frequency, PWM
Lab: software adjusts power to one DC motor

<EdgeInterrupts> 14. Real-Time Systems
Prerequisites: <Timers>
Motivation: edge-triggered interrupts is a good way to service bumper switches on robot
Tools: Oscilloscope, logic analyzer, CCS
Tools: CCS, logic analyzer,
Equipment: bumper switches on robot, LaunchPad
Theory: interrupt driven I/O, Input triggered interrupts
Lab: Input from four switches on robot to detect collision

<ADC> 15. Data Acquisition Systems
Prerequisites: <Timers>
Motivation: the robot uses IR distance sensors to detect walls or other robots
Tools: Oscilloscope, spectrum analyzer, logic analyzer, CCS
Equipment: sensor (IR distance sensor), LaunchPad
Theory: ADC conversion, sampling, periodic interrupts conversion distance to voltage, ADC signal averaging signal to noise ratio, central limit theorem, Nyquist, calibration
Lab: Input distance; detect distance and orientation to wall

<Tach> 16. Tachometer
Prerequisites: <Timers><Motors>
Motivation: The robot can have tachometers to measure wheel rotational speed
Tools: Voltmeter, oscilloscope, CCS
Equipment: Tachometer with digital inputs, DC motor, LaunchPad
Theory: period measurement interrupts
Lab: Measure motor speed

<Control> 17. Control Systems
Prerequisites: <Tach>
Comment: Assume students do not have a lot of control theory. However, they still could implement an incremental and an integral controller.
Motivation: If we have a tachometer or encoder, we can implement a digital controller.
Tools: Voltmeter, current meter, oscilloscope, CCS
Equipment: Robot with tachometer on the motors
Theory: Input capture, incremental control, integral control
Lab: Closed loop DC motor control, spin at constant speed

<UART> 18. Serial communication
Prerequisites: <Debug>
Motivation: Students could use a long USB cable to debug and control the robot in a tethered fashion while the robot is running.
Tools: Voltmeter, oscilloscope, logic analyzer, CCS
Equipment: LaunchPad connected with UART to a PC
Theory: Modulation, encoding, transmission, decoding, error detection, synchronization, FIFO queues
Lab: stream data from robot to PC, build an interpreter so student can manually control the robot from the laptop keyboard.

<BLE> 19. Bluetooth Low Energy
Prerequisites: <UART>
Motivation: Students could use a cell phone to debug and control the robot.
Tools: Logic analyzer, CCS
Equipment: LaunchPad connected with UART to CC2650BP (SNP)
Theory: characteristics, services, advertising
Lab: stream data from microcontroller to phone

<Wi-Fi> 20. Wi-Fi
Prerequisites: <UART>
Motivation: Students stream data from the robot onto a web page.
Tools: Logic analyzer, CCS
Equipment: LaunchPad connected with UART to CC3120 Booster
Theory: UDP TCP DNS, wireless router, creating a web server
Lab: stream data from microcontroller to web server

Module 1

Introduction: Running code on the LaunchPad using CCS

Introduction: Running code on the LaunchPad using CCS

Educational Objectives:

REVIEW Software development methodology
UNDERSTAND How to set up an Integrated Development Environment
EXPLORE The out of box examples
LEARN How to import and export CCS projects
DESIGN, BUILD & TEST A SYSTEM
Understand the debug tools and plug-ins

Prerequisites (None)
- None

Recommended reading materials for students:
- MSP432P401R SimpleLink™ Microcontroller LaunchPad™ Development Kit (MSP‑EXP432P401R) User Guide (SLAU597)
- MSP-EXP432P401R Quick Start Guide (SLAU596)
- MSP432P4xx Technical Reference Manual (SLAU356)
- MSP432P401Rx Datasheet (SLAS826)
- TI Resource Explorer (MSP432 SimpleLink SDK)
- TI SimpleLink Academy

Introduction to the curriculum

In the following modules you will learn about the concepts of robotics in the context of embedded systems. The most important part of the robot will be the main processor or “brain” of the system. The processor will manage the programmable logic of the system and interface with the peripherals for inputs such as sensors and outputs such as motors.

To prepare us to build the robotic system, we will first learn how to master the processor by setting up our hardware development kit and the software development environment used to write the software to control our system.

Software Development

The first step to any embedded development is to set up the software development environment we plan to use once the hardware has been chosen. It is often popular and wise to choose an Integrated Development Environment (IDE). An IDE can have a list of features that aid in the ease or speed of software development. In the hardware context, this could include providing critical debugging information needed to understand the memory usage and performance of the software on the processor.

Code Composer Studio (CCS) is an industry-ready IDE option that is provided by Texas Instruments for use with TI microcontrollers and embedded processors. CCS has many features that make it very capable for professional engineers to develop firmware for real products. It comprises a suite of tools (optimizing C/C++ compiler, source code editor, project build environment, debugger, profiler) used to develop and debug embedded applications. Because it can do so much, it can also be a lot to learn for beginners, but don’t get discouraged as this module will direct you on how to set up CCS so you can go through exercises smoothly as you build your robotic system.

Your code is stored inside of a CCS project. A project can contain many items including your code files, configurations, and other relevant files.

The Project Explorer in CCS shows us the various components used for each project. A **linker** builds a single software system by connecting (linking) software components. In CCS, the **build** command performs both a compilation and a linking.

In an embedded system, the **loader** will program object code into flash on the microcontroller. We place object code in flash ROM because flash is retains its information if power is removed and restored. In CCS, the **Debug** command performs a load operation and starts the debugger.

A **debugger** is a set of hardware and software tools we use to verify system is operating correctly. The two important aspects of a good debugger are control and observability.

A logic analyzer is a tool that will help you debug your circuit. You can view in real time the signals that are being generated on the pins. In this course we will make use of the TExaSdisplay logic analyzer. This is a free tool that works within the MSP432 LaunchPad and uses your PC for display.

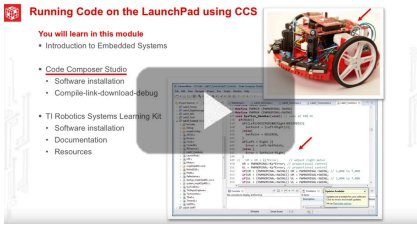
In the lab associated with this module, you will install your copy of Code Composer Studio and test some code examples that are provided for your LaunchPad as a getting started exercise. This will be a good starting point as we familiarize ourselves with the main digital control unit of the explorer robot.

1. TI-RSLK Module 1 Running code on the LaunchPad using CCS

The purpose of this module is to learn software development methodology and understand how to set up an Integrated Development Environment (IDE), to then import and export Code Composer Studio (CCS) projects, as well as critical debugging information to understand the memory usage and performance of the software on the processor.

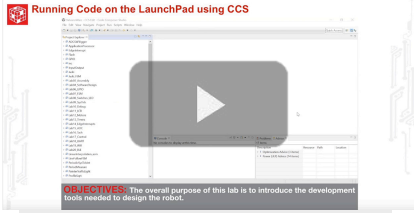
Optionally, download all the curriculum documents for Module 1.

1.1 TI-RSLK Module 1 - Lecture video - Running code on the LaunchPad using CCS
Introduction to embedded systems with CCS and the TI-RSLK software installation

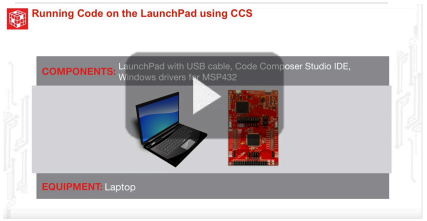


1.2 TI-RSLK Module 1 - Lab video 1.1 - Installing tirslk_maze
You will learn how to set up Code Composer Studio and import the TI Robotics System Learning Kit, Maze Edition.

1.3 TI-RSLK Module 1 - Lab video 1.2 - Getting started with CCS
Lab video accompanying Module 1 - Code Composer Studio Installation and Module 1 lecture and lab from the TI-RSLK curriculum.



1.4 TI-RSLK Module 1 - Lab video 1.3 - Running the TExaS logic analyzer
Additional lab video for Module 1 lab for TI-RSLK curriculum.



1.5 TI-RSLK Module 1 - Lab video 1.4 - Running the TExaS oscilloscope
The overall purpose of this lab is to introduce some of the development tools needed to design your robot.

Module 1

Lab 1: Running code on the LaunchPad using CCS

Lab: Running code on the LaunchPad using CCS

1.0 Objectives

The purpose of this lab is to prepare your workstation to write software that will be loaded on the LaunchPad.

1. You will learn how to install the CCS IDE
2. You will load starter code on the MSP432 LaunchPad.
3. You will learn and practice the debug capability inside the CCS IDE

Good to Know: Using an IDE is an important tool in embedded systems design. This is the crucial first step before interacting with the hardware.

1.1 Getting Started

1.1.1 Software Starter Projects

Look at these four projects:
SineFunction (a simple implementation of sine),
Input_Output (switch input LED output example)
TExaS (example use of logic analyzer and oscilloscope)
UART (serial output to Terminal program, implementing printf)

1.1.2 Student Resources

MSP432P401R SimpleLink™ Microcontroller LaunchPad™ Development Kit (MSP‑EXP432P401R) User Guide (SLAU597)
MSP‑EXP432P401R Quick Start Guide (SLAU596)
MSP432P4xx Technical Reference Manual (SLAU356)
MSP432P401R Datasheet (SLAS826)
TI Resource Explorer (MSP432 SimpleLink SDK)
TI SimpleLink Academy

SimpleLink is a Texas Instruments' umbrella term that includes much of its embedded system produces, such as microcontrollers, wireless, TI RTOS, and IoT.

1.1.3 Reading Materials

TI Resource Explorer, http://dev.ti.com/tirex/
Development Tools-> Integrated Dev. Environ. -> Code Composer Studio

Volume 1 Chapter 1, Sections 2.1, 2.2, and 2.3
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 1.1, 1.2, and 1.3
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

1.1.4 Components needed for this lab

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

1.1.5 Lab equipment needed (none)

1.2 System Design Requirements

Throughout the course you will acquire knowledge that will allow you to build a system that includes mechanical and electrical subsystems. The goal of this first lab is to set up our ability to write firmware for the robot and learn what debugging options are available to troubleshoot the system. In this lab, you will

- Install Code Composer Studio 7.0 or above
- Download and unpack associated files for this course and import the example projects into Code Composer Studio
  1. Data sheet
  2. Software documentation
  3. TExaSdisplay application (scope, logic analyzer)
  4. Example CCS and lab starter projects
- Install the Windows Drivers needed to debug the MSP432 LaunchPad
- Learn the basic steps for software development with CCS
  1. Build (compile)
  2. Debug (download and start debugger)
  3. Run, step, step in, step over, step out
  4. Breakpoint
  5. Observe variables, ports, memory

Note: CCS provides a rich set of debugging tools. Because the robot is an embedded system, we are not concerned with just the software, but rather, we will debug the hardware and software together. As you progress in the course you will continue to discover new features with CCS to help in development and debugging. In general, we can group the techniques into two classifications: control (making the software/hardware system do what you want), and observability (visualizing what the software/hardware system did.)

1.3 Experiment set-up

This lab uses the LaunchPad without any external input or output hardware. All that is needed is your computer that you will use in the course, the MSP432 LaunchPad, and the included USB cable.

1.4 System Development Plan

1.4.1 Installing CCS

You will first need to download the latest CCS version from TI. It is recommended to get at least CCS 7.0 or above to do the work in this course.

http://www.ti.com/tool/ccstudio

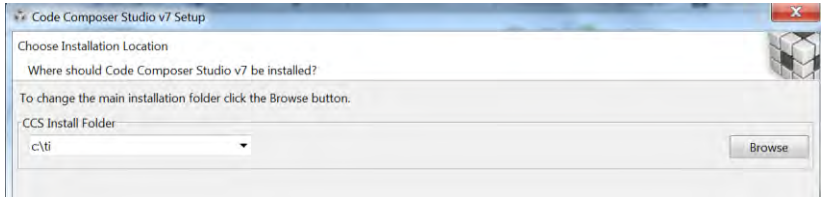
What is the difference between web installer and offline installer? The web installer is a small installation program that you download and execute. You then make your installation selections (device families and features desired) and the installer then downloads and installs only those selected packages. The offline installer is a large package that includes all packages (except for those only available via the CCS App Center). The offline installer is generally only recommended if you have issues with your firewall or anti-virus software blocking the web installer. The offline installer is also useful if you need to install CCS on a machine that does not have internet access. **For this curriculum, you can use either web or offline installer; we suggest using the web installer.**

General tips for installing CCSv7

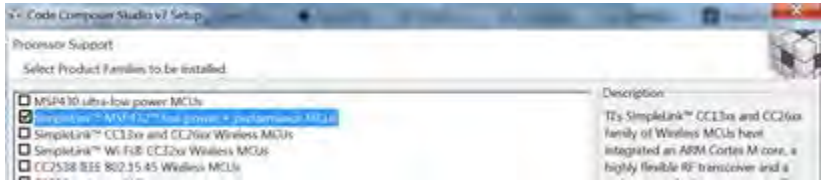
- It is necessary for you to select MSP432 support during installation. MSP432 support includes the device drivers that allow CCS to program and debug LaunchPad software.
- Clean out all prior failed or incomplete installations (by deleting the install directory) before attempting a new one to the same directory. (On the install directory in Windows, use Shift+Del and in Linux and MacOS use rm -Rf <install directory>)
- If you plan to install two versions side-by-side, **always** use different workspaces. Sharing a workspace between two versions may cause severe impact in project building and debugging.
- Disable anti-virus (certain anti-virus software is known to cause problems). If it cannot be disabled, try the offline installer instead of web installer: Download CCS
- Ensure that your **Username** does not have any non-ASCII characters, and that you are installing CCS to a directory that does not have any non-ASCII characters. A temporary directory using the **Username** is created during installation. Eclipse is unable to handle non-ASCII characters. If your **Username** does have non-ASCII characters, please create a temporary admin user for installing CCS.

1.4.2 Running the CCS installer

Begin the installation process after downloading the latest version of CCS. By default it will ask you to install under a ti folder, which is recommended.



During the initial setup please make sure that you select processor support for SimpleLink MSP432 MCUs. The processor support matches our MSP432P401R LaunchPad development kit. Installing other processor support is optional but this course will only use the MSP432.



Under Debug Probe support selection, make sure that the default “TI XDS Debug Probe Support” is selected. This is the debugger used on the LaunchPad development kit. The other options are for external debuggers, but these debuggers will not be used in this course.

Lab: Running code on the LaunchPad using CCS

Click finish and your installation should proceed to completion. When completed you can open CCS and select your workspace. The default workspace is recommended other projects but for this course you will create a custom workspace called **tirslk_maze,** as described in the next section.

1.4.3 Import tirslk_maze

We are going to import all the curriculum project folders into CCS for our next step, creating one workspace for the entire course.

tirslk_maze is a set of software components that includes many (40) CCS example projects, html documentation, data sheets, and a Windows application called **TExaSdisplay**. Some of the example projects run out of the box and are intended to illustrate various functionalities of the MSP432. However, some of the projects have names beginning with "Lab", and these are starter projects for your labs. You will be developing code in these projects as part of the lab assignments in this curriculum. Furthermore, the folder **inc** has files you develop in one lab that will be used in subsequent labs. The steps to install **tirslk_maze** are

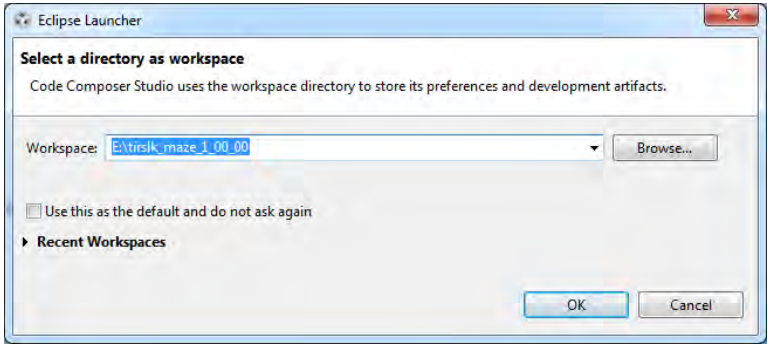
Step 1: Download the archive file (zip file)
<download zip file>

Step 2: Extract the zip to a file location you want the projects to reside. Preferably an easy to find location on your computer. Once unzipped and compiled, the **tirslk_maze_1_00_00** folder will expand to about 200 MB. In the subsequent figures you can see I extracted it to E:\

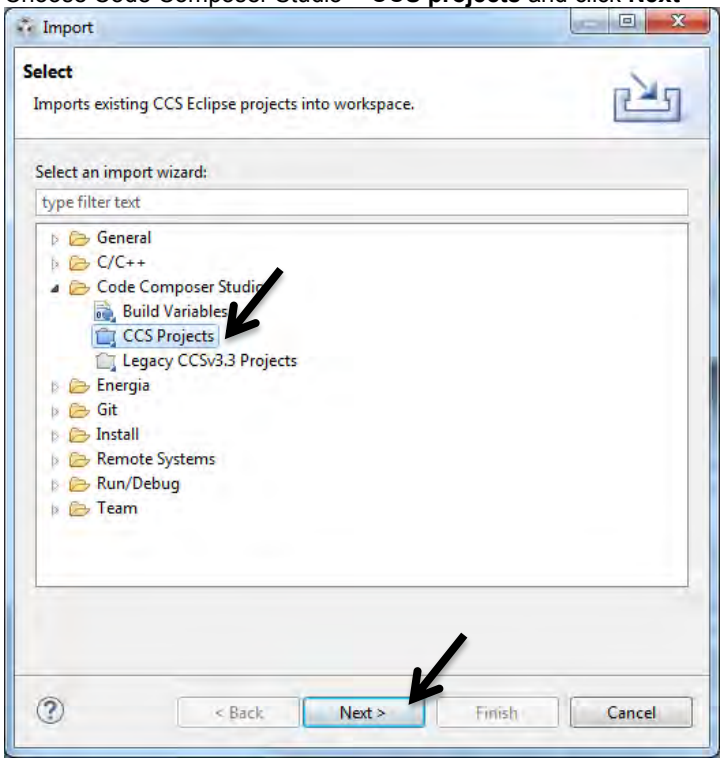
Step 3: Open the software documentation by double-clicking on the file
tirslk_maze_1_00_00_Software_Documentation.html
This documentation includes software provided to you as examples and software you will write as part of the lab sequence.

Step 4: Open the datasheets folder. In this directory you will find descriptions of the hardware components used in this curriculum. We suggest you begin with these two reference manuals
Meet the LaunchPad, slau596.pdf
MSP432P4xx Technical Reference Manual, slau356f.pdf

Step 5: Start CCS. The simplest approach to setting up the software for this course is to use the unzipped folder from step 2 as your workspace. You need to switch to this workspace using **File > Switch Workspace**. Browse to the unzipped folder from step 2 and select OK.

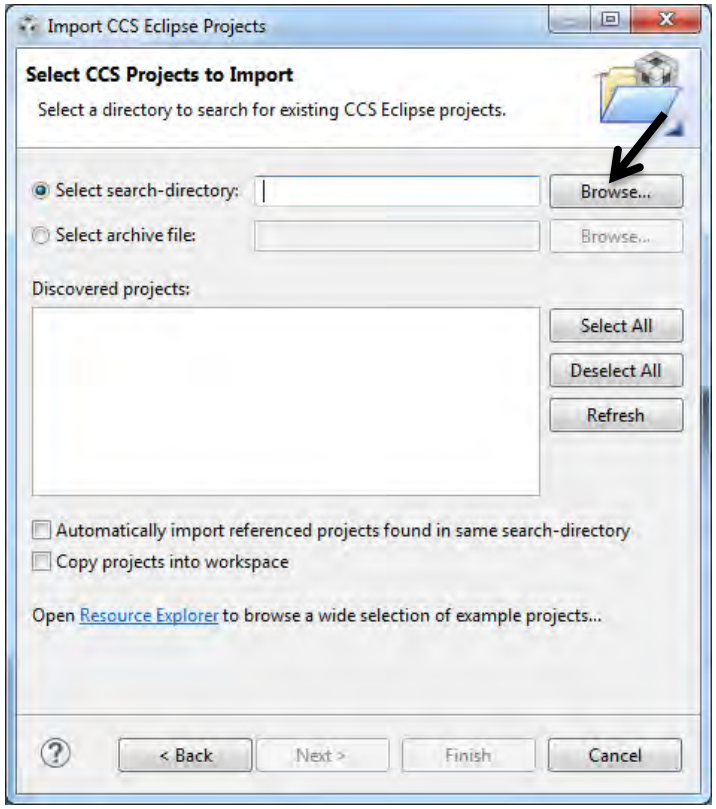


Step 6: Import all the projects into CCS. From the menu bar, click
File > Import…
Choose Code Composer Studio > **CCS projects** and click **Next>**



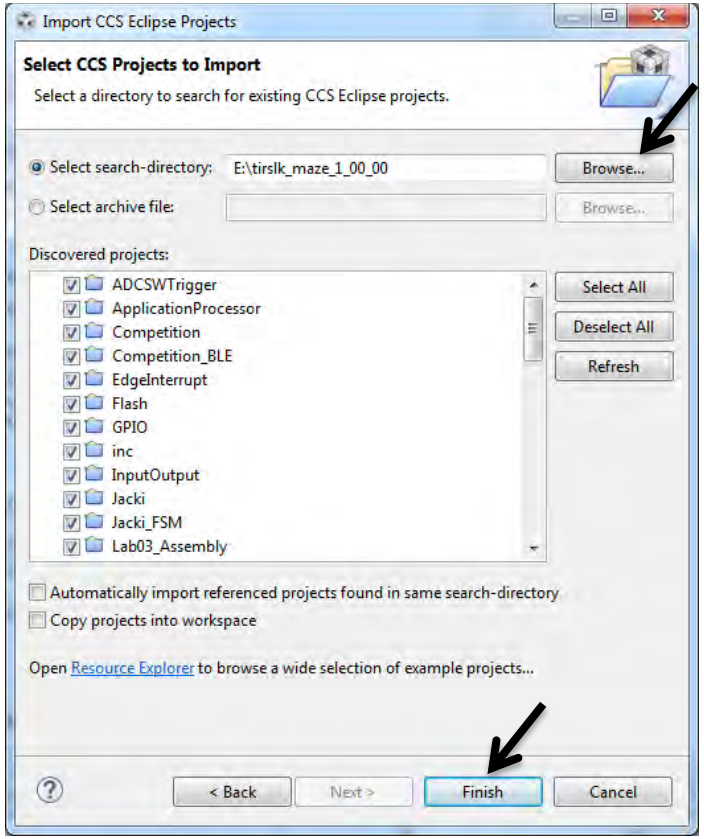
Lab: Running code on the LaunchPad using CCS

Select search-directory and click the **Browse…** option, and find the unzipped **tirslk_maze_1_00_00** folder that you created in step 2.



CCS should discover many projects inside the **tirslk_maze_1_00_00** folder.

Click **Select All** (do not check **Automatic import** or **Copy projects** options). This will have CCS reference the project from the original location and preserve the original directory structure required to build. Click Finish



Now your projects for the course are imported and visible in the project explorer. We are now set up with CCS!

Step 7: Notice over 40 projects in the project explorer. All but one of the projects will be used in this curriculum. Click on the **inc** folder within Project explorer and notice the files within the folder. We will not be using the **inc** project for any code development. The **inc** folder contains software that will be shared between projects throughout the curriculum. The **inc** project was created for the sole purpose of making it easy for you to open the files from the project explorer. The **inc** files, listed in Table 1, are completely written and available as example code for the MSP432. On the other hand, you are required to complete the **inc** files that are listed in Table 2 as part of the lab assignments. For both sets of files the software documentation explains what the functions are and how to use them.

Lab: Running code on the LaunchPad using CCS

Header	Code	Purpose
AP.h	AP.c	Application Processor, BLE
Clock.h	Clock.c	Sets bus clock to 48 MHz
CortexM.h	Cortex.c	Enable and disable interrupts
FlashProgram.h	FlashProgram.c	Erase and program flash
GPIO.h	GPIO.c	Digital I/O, CC2650 BLE
LaunchPad.h	LaunchPad.c	LaunchPad LEDs / switches
LPF.h	LPF.c	Low pass filters
SysTick.h	SysTick.c	24-bit system timer
SysTickInt.h	SysTickInt.c	Periodic interrupt
TA0InputCapture.h	TA0InputCapture.c	Period measurement
TA2InputCapture.h	TA2InputCapture.c	Period measurement
Tachometer.h	Tachometer.c	Tachometer interface
TExaS.h	TExaS.c	Oscilloscope, logic analyzer
Timer32.h	Timer32.c	32-bit periodic interrupt
TimerA0.h	TimerA0.c	16-bit periodic interrupt
TimerA2.h	TimerA2.c	16-bit periodic interrupt
UART.h	UART.c	Serial port
Ultrasound.h	Ultrasound.c	Ultrasonic sensor interface

Table 1. Shared files you can use. I.e., these files are complete and functional.

Header	Code	Purpose (Lab)
ADC14.h	ADC14.c	Analog to digital conv. (15)
Bump.h	Bump.c	Bump sensors (10)
BumpInt.h	BumpInt.c	Interrupting sensors (14)
	convert.asm	Assembly functions (3)
IRDistance.h	IRDistance.c	Distance conversions (15)
Motor.h	Motor.c	Motor interface (13)
MotorSimple.h	MotorSimple.c	Simple motor interface (12)
Nokia5110.h	Nokia5110.c	LCD interface (11)
PWM.h	PWM.c	Pulse width modulation (13)
Reflectance.h	Reflectance.c	Line sensor (6 and 10)
TA3InputCapture.h	TA3InputCapture.c	Input capture, tachometer (16)
TimerA1.h	TimerA1.c	Periodic interrupt (13)
UART1.h	UART1.c	Interrupting serial port (18)

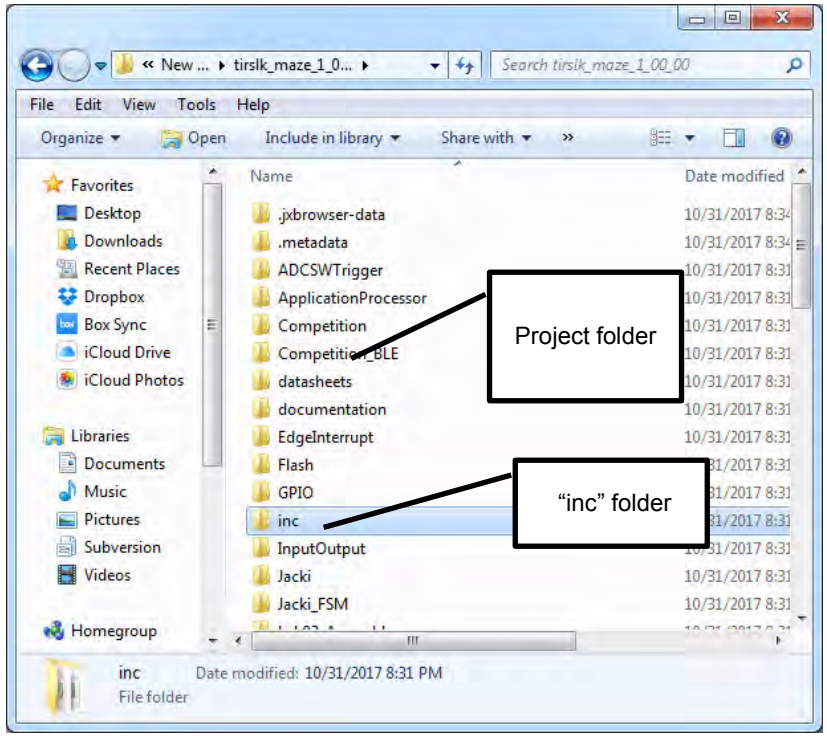
Table 2. Shared files you will need to complete. I.e., you need to complete the functions in these files in order for them to operate properly.

1.4.4 Project structure

To better understand this course we need to explain the project structure inside the **tirslk_maze_1_00_00** folder. **tirslk_maze_1_00_00** is an archive of the course projects that a student can unpack, compile and build in CCS. There are more than 40 projects with the same structure.

- CCS 7.x, C99 language, **doxygen** documentation
- Configured for the MSP432P401R LaunchPad
- No use of TI libraries or any external libraries
- Just C code (there is one **Solution.obj** in Lab4)

There is one folder with shared C code called "**inc**" that contains files used in multiple projects. For example, **bump.c** and **bump.h** are in the **inc** folder. Whenever a project wishes to use one of these shared files, the code file (e.g., **bump.c**) is added to the project (linked) and the header file is included using #include (e.g., **#include "..\inc\bump.h"**)



Note that projects with "Lab" in the name are intended as starter projects for each lab. Other projects are examples. The projects that begin with "Competition" can be used to develop high-level code without developing all the low-level I/O driver code.

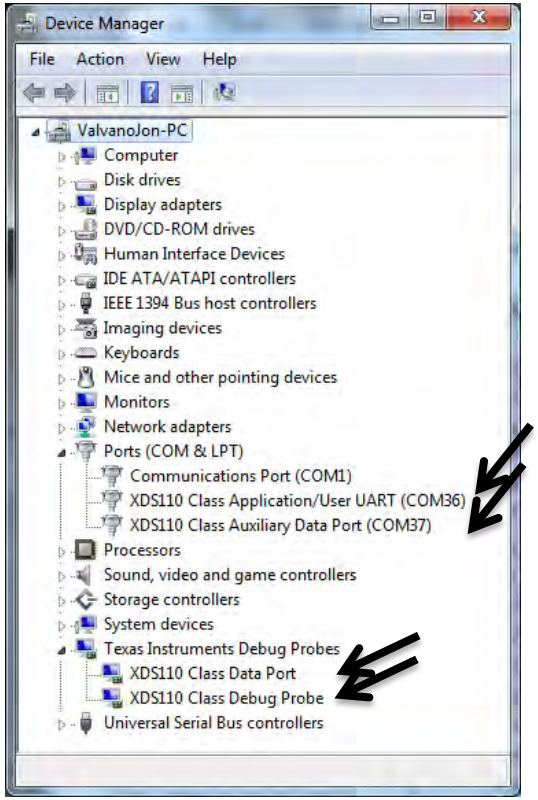
Lab: Running code on the LaunchPad using CCS

1.4.5 Installing OS drivers for the LaunchPad

Drivers are OS software that allow CCS to communicate with the XDS110 debugger on the LaunchPad.

The first step to installing drivers is to plug the MSP432 LaunchPad into the PC using the USB cable. Some LEDs on the LaunchPad should light up. When you plug your LaunchPad into a USB port on your computer, the operating system will attempt to load drivers. If you selected MSP432 support during CCS installation, then the operating system should automatically find the drivers. On Windows there will be four drivers in the device manager associated with the LaunchPad.

Notice there are two COM ports. We will exclusively be using the first one (the one with the lower number).



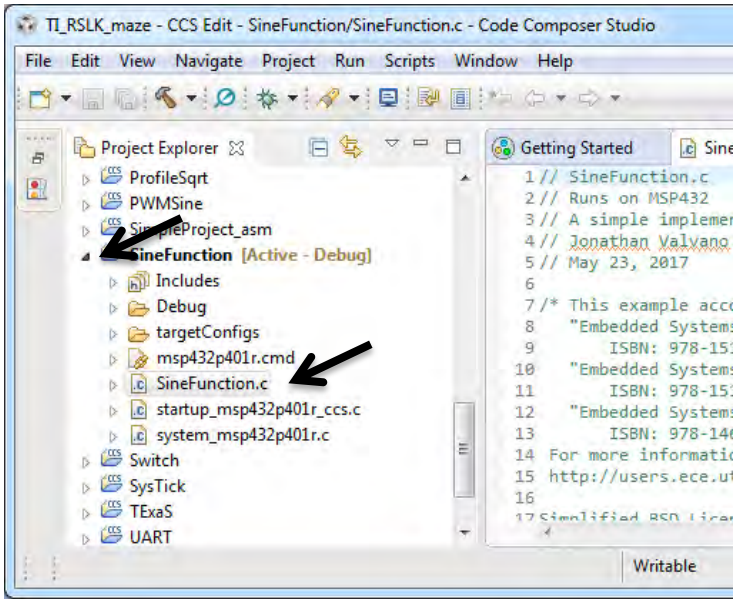
1.4.6 Run a simple example on the LaunchPad

Key Objectives

- Observe source code
- Edit-compile-link-download-debug cycle
- Step in, step over, step out
- Observing local and global variables

SineFunction is a very simple software project that performs no input/output. It calculates a **y=sin(x)** using a cubic approximation and fixed-point math. It fills an array with results. You can use the project to learn how to build (compile), debug (download and start the debugger), run, halt, and observe the array. You should also reset the processor, set a breakpoint, run until the breakpoint, and then single step (step in, step out, and step over).

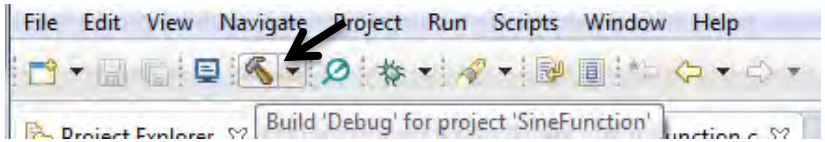
1) Click on the **SineFunction** project, and open the view of the files in that project. Double click on **SineFunction.c** to see the source code.



Note: Make sure the desired project is in context (in bold) before building or debugging. Notice in the above figure, the Project Explorer bolds the project and specifies "**[Active-Debug]**"

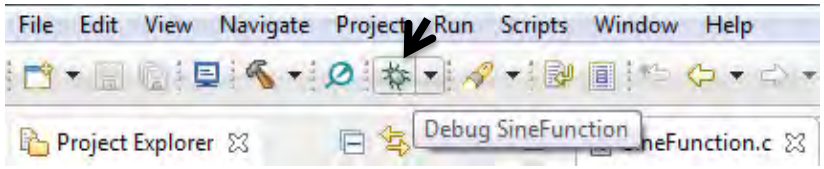
Lab: Running code on the LaunchPad using CCS

2) With the **SineFunction** project selected [Active-Debug], click Build



There should be no errors.

3) With the **SineFunction** project selected [Active-Debug], click **Debug**



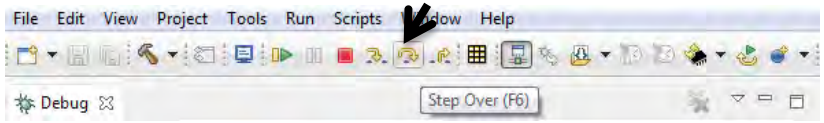
Note: When you debug on your LaunchPad for the first time it may prompt you to update the firmware. This step is recommended.

The debug operation causes several actions to be done automatically

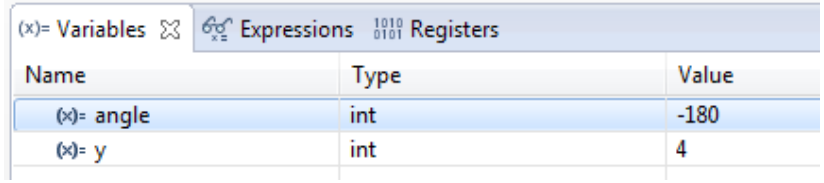
- Prompt to save source files
- Build the project (incrementally)
- Start the debugger (CCS will switch to the CCS Debug perspective)
- Connect CCS to the target
- Load (flash) the program on the target
- Run to main

4) Once the flash is erased, and the image of this project is loaded, you will see the green triangle appear. That is the run icon, don't click run yet, but seeing this icon means the system is ready to debug

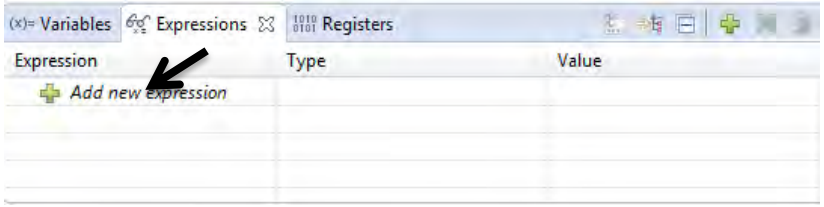
5) Single step the program by executing **Step Over** icon multiple times



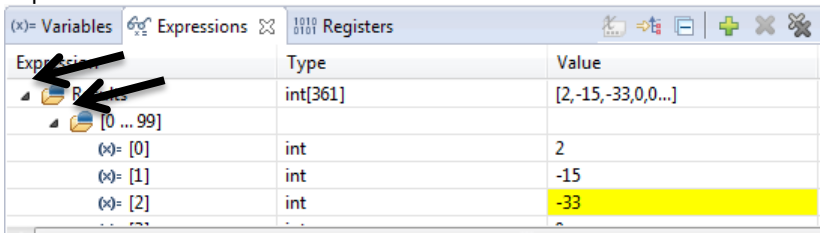
6) Observe the local variables in the **Variables** window



7) Observe global variables in the Expressions window. Type **Results** in the "Add new expression" field and hit <enter>



Expand the Results field to see its data.



Step over executes one line of C. If that line has a function, step over will execute the entire function. You can also experiment with **Step in** (which executes one line of C, and if that line has a function, it will step into that function. If you have stepped into a function, **Step return** will complete that function and stop at the spot the function was called.

You should also experiment the **Resume**, **Suspend**, and **Reset** commands.

8) To **halt** the debugger and terminate execution, click the Terminate icon

Lab: Running code on the LaunchPad using CCS

1.4.7 Run the Input_Output example on the LaunchPad

Key Objectives

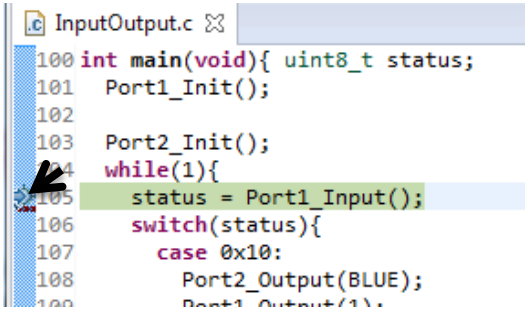
- Observe I/O ports on the MSP432
- Interact with hardware on the LaunchPad
- Setting and clearing breakpoints

Input_Output is a simple project that showcases some the features of the LaunchPad. For example, it will input from the two switches on the LaunchPad and output to the LED. Follow the same steps 1, 2, 3, and 4 as you did to compile and load this project onto MSP432 LaunchPad.

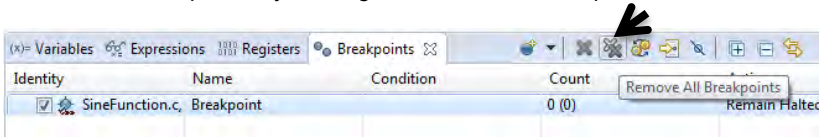
1) Run the project and interact with the two switches on the LaunchPad. You should observe this simple behavior

No switches	No LEDs on
Just SW1	Red LED is on, color LED is blue
Just SW2	Red LED is on, color LED is red
Both SW1,SW2	Red LED is on, color LED is blue+red=purple

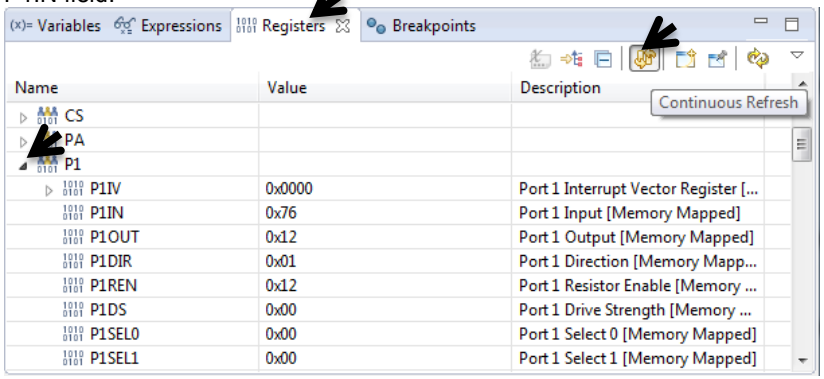
2) Set a breakpoint at the line **status = Port1_Input();** To place a **breakpoint**, click on a line at which you want it to stop, right click and add hardware breakpoint. When you start the program it will run to the breakpoint and stop. The following figure shows the debugger halted at the breakpoint.



Remove all breakpoints by clicking the icon in the breakpoint window



3) Observe the I/O Port registers. First select the **Registers** tab, then select **P1** (I/O Port 1). Activate the Continuously Refresh mode. Run the program and touch the two switches on the LaunchPad. You will see the port input data in the P1IN field.



1.4.8 Run the TExaSdisplay logic analyzer

Key Objectives

- Introduce TExaSdisplay in logic analyzer mode
- Observe digital signals on the LaunchPad

TExaSdisplay is a Windows application that does not require a separate installation. You should see the **TExaSdisplay.exe** executable within the **tirslk_maze_1_00_00** folder. To start the application, you simply double-click **TExaSdisplay.exe** executable file.

If you have access to a real logic analyzer, you should use it, and therefore can skip this section. If you do not have access to a real logic analyzer, then TExaS provides a no-cost, simple option. TExaS has these specifications:

- Up to 7 digital channels
- 10 kHz sampling (you can adjust the display but sampling is fixed)
- Runs in background alongside your software
- Data streamed through USB cable from MSP432 to PC

The first step is to activate the TExaS project, and open the **TExaSmain.c** file. There are three main programs in this project. Edit the **LogicAnalyzerMain** function so it is called main, and edit the other main to be Lab2main. Notice the MSP432 will be running at 48 MHz and the logic analyzer is configured to display Port 1. When it is running the seven bits of P1.6 – P1.0 will be streamed to the PC at 10 kHz. The logic analyzer works whether the pin is an input or output. In this example, P1.0 is an output (to the red LED) and P1.1/P1.4 are inputs from

Lab: Running code on the LaunchPad using CCS

the two LaunchPad switches. We will talk about I/O in great detail in subsequent chapters, but for now let's focus on how the logic analyzer measures P1.4, P1.1, P1.0 by sending the digital information from the MSP432 to the PC via the USB cable.

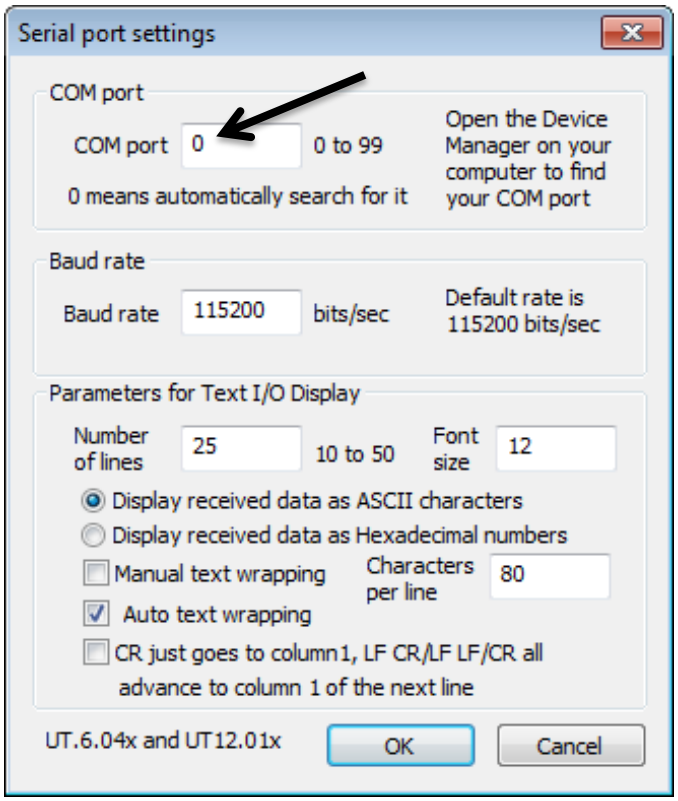
```
int LogicAnalyzerMain(void){
uint32_t status,delay,data;
  Clock_Init48MHz(); // makes bus clock 48 MHz
  LaunchPad_Init();  // use buttons to step through frequencies
  TExaS_Init(LOGICANALYZER_P1);
  data = 0;
  while(1){
    status = LaunchPad_Input();
    switch(status){    // negative logic on P1.1 and P1.4
      case 0x00: delay=1000; break;  // neither switch pressed
      case 0x01: delay=2000; break;  // SW2 pressed
      case 0x02: delay=3000; break;  // SW1 pressed
      case 0x03: delay=4000; break;  // both switches pressed
    }
    Clock_Delay1us(delay);
    data = data ^0x01;
    LaunchPad_LED(data); // toggle red LED
  }
}
```

You can see the various options for the logic analyzer by looking in the **TExaS.h** header file. These are the choices you have when configuring the TExaS.

```
enum TExaSmode{
  SCOPE,          //8-bit oscilloscope on J3.26/P4.4/A9
  LOGICANALYZER,    //7-bit logic analyzer
  LOGICANALYZER_P1,  //  7-bit logic analyzer on P1.6-P1.0
  LOGICANALYZER_P2,  //  7-bit logic analyzer on P2.6-P2.0
  LOGICANALYZER_P3,  //  7-bit logic analyzer on P3.6-P3.0
  LOGICANALYZER_P4,  //  7-bit logic analyzer on P4.6-P4.0
  LOGICANALYZER_P5,  //  7-bit logic analyzer on P5.6-P5.0
  LOGICANALYZER_P6,  //  7-bit logic analyzer on P6.6-P6.0
  LOGICANALYZER_P7,  //  7-bit logic analyzer on P7.6-P7.0
  LOGICANALYZER_P8,  //  7-bit logic analyzer on P8.6-P8.0
  LOGICANALYZER_P9,  //  7-bit logic analyzer on P9.6-P9.0
  LOGICANALYZER_P10,  //  7-bit logic analyzer on P10.6-P10.0
  LOGICANALYZER_P4_765432,  //  6-bit logic analyzer on P4.7-P4.2
  LOGICANALYZER_P4_765320,  //  6-bit logic analyzer on P4.7-5,3-2,0
  LOGICANALYZER_P2_7654  //  4-bit logic analyzer on P2.7-P2.4
};
```

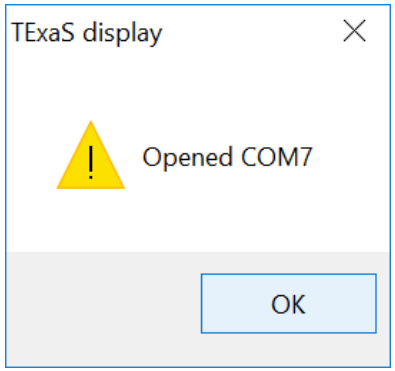
Build (compile), debug (erase flash, program flash with object code), and run the project. The red LED flashes, and you can change the rate of flashing by pushing the two switches.

Start **TExaSdisplay** and execute COM->Settings. You can enter the COM port number (which you can find from your device manager), or you can leave the field at "0", which means start at 1 and search for a COM port that will open. The baud rate is always 115200 bits/sec in this class, but for other situations you might need to set the baud rate. The other parameters in this dialog configure the look and feel of the text window, when using **TExaSdisplay** as a terminal application.

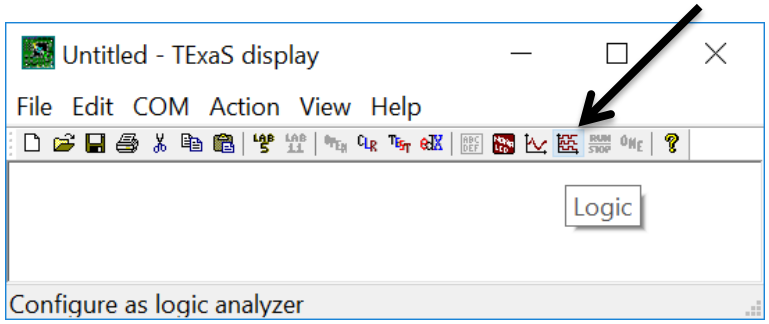


To connect **TExaSdisplay** with the MSP432 serial port, you click the **Open** tool button or execute the command **COM -> Open Port** (F4). On this computer, the MSP432 LaunchPad was found as COM7.

Lab: Running code on the LaunchPad using CCS



To run **TExaSdisplay** in logic analyzer mode, you click the Logic Analyzer tool button, or execute **View -> Logic Analyzer**



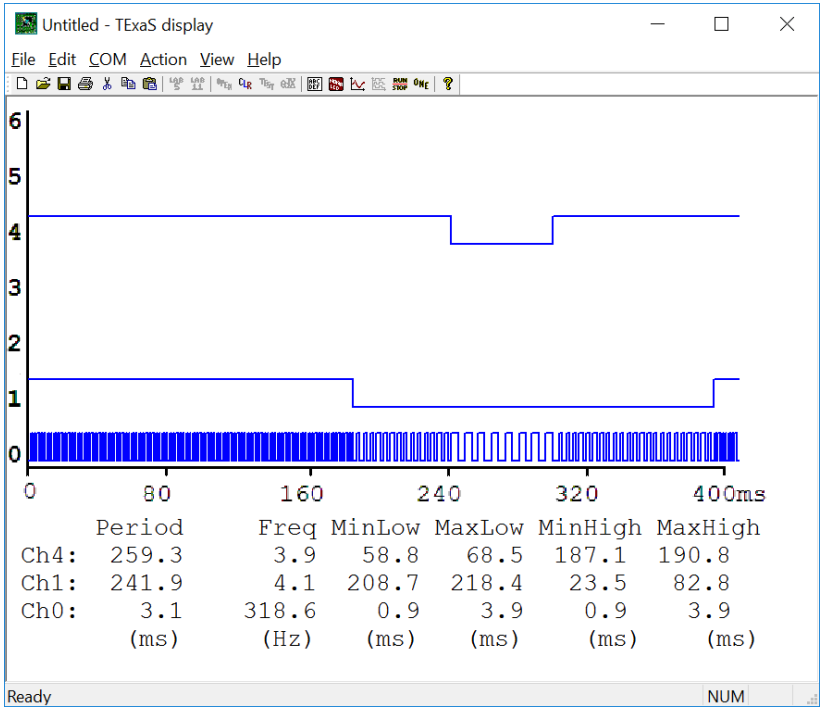
The logic analyzer always sends 7 bits at 10 kHz, but you can choose which ones to plot on the display. Execute **View -> Logic Analyzer Configuration** and disable pins 6,5,3,2 (because they have no value in this example). Specify a rising edge trigger in bit 0. The analyzer takes multiple readings and lines up the traces using the trigger. Using a trigger means one of the traces will not jitter around if the data are rapidly changing.

An **edge trigger** means the analyzer will search the incoming data stream for an edge on that pin, plotting the edge at a fixed place on the screen. The analyzer takes multiple readings. With a rising edge trigger active the rising edge is placed at the same location of the display. Using a trigger means one of the traces will not jitter around if the data are rapidly changing.

When your MSP432 program is running, you will be able to see digital data versus time. There are four commands to control the display

View->Slower F6 will increase the range of times displayed
View->Faster F7 will decrease the range of times displayed
View->Pause/Run F8 will stop/start the display
View->Single F9 will display one sweep and stop

You will see the three pins (SW1, SW2, and LED) plotted versus time. Push the two switches to observe the behavior that the switches affect the frequency of the oscillations on P1.0.



Lab: Running code on the LaunchPad using CCS

1.5 Troubleshooting

A project doesn't compile:

- Try a different project. All the projects in **tirslk_maze** should compile. If none of the **tirslk_maze** compile, then try reinstalling CCS and **tirslk_maze**.
- If other projects in **tirslk_maze** compile, but a project you have edited does not compile, it is possible you have introduced errors. Follow the error codes in the **problems** window. Remember to start with a project that compiles, make only a few changes, and then compile it. This way when it doesn't compile, there are only a few places to look for the error.

The debug command can't erase/download/run:

- Make sure the build step occurred without error.
- Check the device manager to make sure the proper drivers are installed for the LaunchPad board.
- Make sure the LaunchPad power is connected to the PC.
- Try another USB port.
- Try another micro USB cable

1.6 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.

- What are components of a project on CCS?
- What are the steps involved in software design/test?
- What are breakpoints? How do I set them up? How do I use breakpoints to debug?
- What does it mean to step in, step over, and step out?
- What are some of the ways to observe intermediate results during software debugging?
- What is a logic analyzer? What is an oscilloscope?

1.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. Additional challenges are not required to complete the course.

- Run the UART example (Appendix A2)
- Run the TExaS oscilloscope example (Appendix A3)
- Load a project with resource explorer (Appendix A4)
- Run an energy trace on a system (Appendix A6)

1.8 Which modules are next?

Now that we have started, there are two paths forward. The hardware path involves learning about electronics and building the robot:

Module 2 - Study voltage current power and the batteries
Module 5 - Robot construction, including battery and voltage regulation
Module 12 - Interfacing the motors and wheels

The software path involves developing programming and debugging skills:

Module 3 - Introduce the Cortex M processor
Module 4 - Introduce the process of software design
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

1.9 Things you should have learned

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

- Install CCS
- Import projects for the **tirslk_maze** curriculum
- Compile and run a program on the MSP432 LaunchPad
- Use of Debug mode in CCS

Module 2

Introduction: Voltage, Current and Power

Introduction: Voltage, Current and Power

Educational Objectives:

REVIEW Electric circuits with resistors, capacitors and inductors
UNDERSTAND Voltage, current, and power
EXPLORE Behavior of resistors, capacitors, and LEDs
LEARN How to use an oscilloscope
MEASURE Voltage and current in resistors, capacitors and LED

Prerequisites (Module 1)
- Running code on the LaunchPad (Module 1)

Recommended reading materials for students:
- Volume 1 Section 1.1,
  Embedded Systems: Introduction to the MSP432 Microcontroller
  ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or
- Volume 2 Sections 8.1, 8.3.1, and 9.1
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

This module serves as a brief overview of the electrical engineering terms used in the circuits of this class. As a prerequisite of this course, we expect the students to have basic understanding of resistors, capacitors, and inductors. The electrical circuits for the robot explorer are either given or very simple, so this course does not entail circuit design. However, students will need to understand voltage, current, and power as they apply to these circuits. This module presents formal definitions of voltage, current, and power. The lab provides a simple means to discover these parameters for resistors, capacitors, and LEDs.

Current (I) is defined as the movement of electrons. Current is directional and measured at one point as the number of electrons travelling per second. Current has amplitude and a direction. Because electrons are negatively charged, if the electrons are moving to the left, we define current as flowing to the right.

Voltage (V) is an electrical term representing the potential difference between two points. The units of voltage are volts (V), and it is always measured as a difference. Voltage is the electromotive force or potential to produce current.

Another important parameter occurring when current flows through a device is **power**. The power (P in watts) dissipated in a device can be calculated as the product of voltage (V in volts) times current (I in amps). Interestingly, although voltage has a polarity (+ and –) and current has a direction, power has neither a polarity nor a direction. Resistors, capacitors, LEDs, and motors dissipate power in different ways.

The **energy** (E in joules) stored in a battery can be calculated from voltage (V in volts), current (I in amps), and time (t in seconds). Energy has neither polarity nor direction.

An **oscilloscope**, or scope, graphically displays information about an electronic circuit, where the voltage amplitude versus time is displayed. A scope has one or more channels, with many ways to trigger or capture data.
A scope is particularly useful when interfacing sensors and actuators to the robot explorer.

A **signal generator** is a device that creates a voltage versus time output. Example waveform shapes are square waves, pulses and sine waves. Some generators allow you to control frequency and voltages of these waves.

In the lab associated with this module, you will build some simple circuit with resistors, capacitors, and LEDs. Using a voltmeter and current meter, you will study the steady state response (direct current, **DC**) of resistor and LED circuits. In this way, you can discover voltage, current and power. Using a signal generator and an oscilloscope, you will study the transient response (alternating current, **AC**) of the resistor/capacitor circuits.

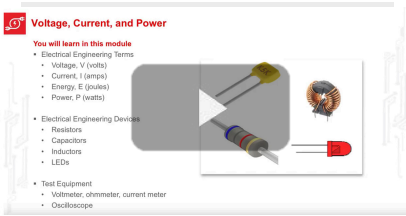
2. TI-RSLK Module 2 – Voltage, current, and power

The purpose of this course is to review basic electronic components and the electrical properties needed to interface sensors and actuators to a microcontroller. You will learn how to measure reactance of a capacitor and use your project to measure current and voltage. The electrical properties of the capacitor will help you design circuits that “filter” or remove noise from your robot.

Optionally, download all the curriculum documents for Module 2.

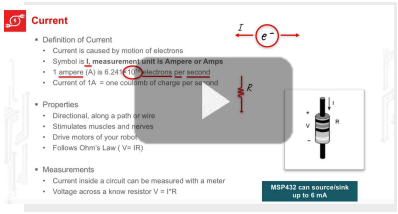
2.1 **TI-RSLK - Module 2 - Lecture video - Voltage, current and power**

Module 2: Voltage Current and Power will cover resistors, capacitors and LEDs.



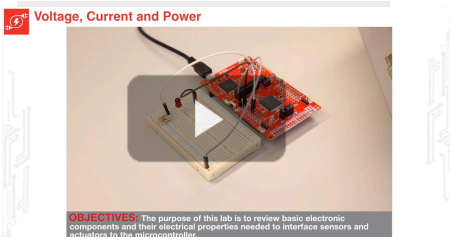
2.2 **TI-RSLK Module 2 - Lab video 2.1 - Measuring the reactance of a capacitor**

The purpose of this lab is to review basic electronics needed to interface sensors and actuators to the microcontroller.



2.3 **TI-RSLK Module 2 - Lab video 2.2 - Measure LED (I,V) response curve**

In this particular portion of the lab, you will measure voltage and current across the LED.



Module 2

Lab 2: Voltage, Current and Power

Lab 2: Voltage, Current and Power

2.0 Objectives

The purpose of this lab is to review basic electronics needed to interface sensors and actuators to the microcontroller.

1. You will learn about voltage, current, and power.
2. You will perform experiments with resistors, capacitors, and LEDs.
3. You will discover both DC and AC responses of circuits.
4. You will use a voltmeter to perform the DC analysis
5. You will use a signal generator and oscilloscope to perform the AC analysis

Good to Know: When interfacing any two physical devices (e.g., sensors to the microcontroller, or microcontroller to an actuator), it is important to manage the voltage and current levels between the devices. Furthermore, when dealing with time-varying signals, resistance, capacitance, and inductance all affect the behavior.

2.1 Getting Started

2.1.1 Software Starter Projects

If you do not have access to a real signal generator and oscilloscope, you can run this project to activate the ***TExaS scope: TExaS.*** The project implements an oscilloscope on pin P4.4. It also creates a square wave out on P4.5. You can adjust the frequency of the squarewave by changing the FREQ constant in the **main.c** file.

2.1.2 Student Resources

CarbonFilmResistor.pdf	Data sheet for resistor
CeramicCapacitor.pdf	Data sheet for ceramic capacitor
LTL-10223W.pdf	Data sheet for 10 mA red LED
HLMP-4700.pdf	Data sheet for 2 mA red LED

2.1.3 Reading Materials

Volume 1 Section 1.7, Chapter 3, and Section 5.3
Embedded Systems: Introduction to the MSP432 Microcontroller",

Volume 2 Sections 1.1, 2.1, and 2.5
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller",

2.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	Red 10mA 5mm,	Lite-On	LTL-10223W
1	Red 2mA 5mm,	Broadcom	HLMP-4700
1	Carbon 1/6W, 5%, 220 ohms	Yageo	CFR-12JB-220R
1	Carbon 1/6W, 5%, 470 ohms	Yageo	CFR-12JB-470R
1	Carbon 1/6W, 5%, 22k	Yageo	CFR-12JB-22K
1	Carbon 1/6W, 5%, 33k	Yageo	CFR-12JB-33K
1	Ceramic, Z5U, -20/+80%, 0.47 µF	Kemet	C320C474M5U5TA

2.1.5 Lab equipment needed

Oscilloscope (one or two channels at least 10 kHz sampling)
Signal generator (10 Hz to 1000 Hz waveforms)

Lab 2: Voltage, Current and Power

2.2 System Design Requirements

The goal of this lab is to measure the current through and voltage across resistors, capacitors and LEDs.

For resistors, you will observe Ohm's Law:

$$V/I = R$$

When experimenting with resistors, you will work with DC voltages (e.g., constant, not time-varying). Although you will experiment with DC voltages, Ohm's Law will also apply to AC responses as well.

For capacitors, you will observe the reactance of a capacitor:

$$V/I = X = 1/(2\pi fC)$$

where V is the AC amplitude of the voltage, and I is the AC amplitude of the current. The reactance of the capacitor at DC will be infinite (DC means f=0). So at DC, the capacitor will not conduct any current. Thus, when experimenting with capacitors, you will work with AC voltages.

LEDs are semiconductor devices with a nonlinear voltage/current response. Your goal is to experimentally observe this nonlinear response. Just like resistors, you will study the DC voltage-current response of LEDs. This nonlinear response will also apply to AC voltages.

2.3 Experiment set-up

If you have an actual signal generator and scope, you should use them. However, if you do not have access to a signal generator and scope, you can use the ***TExaS*** project running on the LaunchPad together with the ***TExaSdisplay*** application running on the PC.

To use the TExaS oscilloscope, connect the TI's LaunchPad development board to a USB port on your PC, build and debug the TExaS project.
Notice the initialization is performed with SCOPE as the mode.

Warning: Ensure the voltages you are analyzing remain between 0 the 3.3V.

The TExaS project software must be running on the MSP432, and the LaunchPad must be connected to the PC via its USB cable. Connect the signal you wish to measure to P4.4.

To observe the signal perform the following tasks:

1. Run the **TExaSdisplay** application on your PC
2. Execute **COM->OpenNextPort** until the MSP432 is connected
3. Execute **View->Oscilloscope** to see the scope

Within the **TExaSdisplay** application, you can press F6/F7 to adjust the time scale, and press the Up/Down arrows to adjust the trigger threshold. The sampling rate is fixed at 10 kHz, the range is 0 to 3.3V, and the precision is 8 bits.

2.4 System Development Plan

2.4.1 Ohm's Law

You will use four different resistors {e.g., 220, 470, 22k and 33k}. Any four resistors in the 220 to 33k range will suffice. If you have an ohmmeter, measure the actual resistance of the four resistors. If you do not have an ohmmeter, you can assume the resistance value is as defined by the resistance color code.

Using the four resistors, build four circuits similar to Figure 1, using the LaunchPad 3.3V as the power source (e.g., 220+22k, 470+22k, 470+33k, and 22k+33k). You must connect both 3.3V and ground to the circuit.

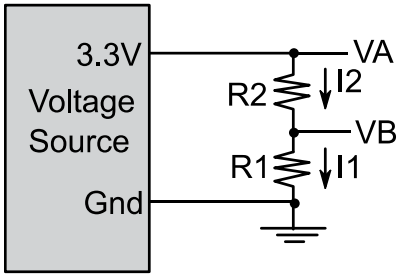


Figure 1. Resistance circuit.

Lab 2: Voltage, Current and Power

For each circuit, measure VA and VB. The voltage across R2 is VA-VB. Calculate

I2 = (VA-VB)/R2
I1 = VB/R1

Ohm's Law is true if the calculation of I2 equals I1. If you have a current meter, you could also compare the calculation of I1 and I2 to the actual measured current.

For each circuit calculate the power dissipated in each resistor

P2 = (VA-VB)*I2
P1 = VB*I1

In summary, fill in the fields of Table 1.

R2	VA-VB	I2	P2	R1	VB	I1	P1
220				22k			
470				22k			
470				33k			
22k				33k			

Table 1. Experimental verification of Ohm's Law

2.4.2 Reactance of a Capacitor

You will use the circuit shown in Figure 2 to study the behavior of capacitors. As mentioned earlier, we will study the circuit in the AC, or time-varying mode. You will use a signal generator to create an AC signal on VA, and use an oscilloscope to measure the AC signal across VA-VB and on VB. It is best if the shape of the AC signal is sinusoidal, but we could perform the lab with other waveform shapes, such as square wave.

This circuit is most interesting at its cutoff frequency defined below:

$f_c = 1/(2\pi RC)$

For example, if R=470 ohms and C is 0.47µF, the cutoff frequency is 720 Hz. You may perform this analysis with any values of R and C with a cutoff frequency between 100 and 1000 Hz.

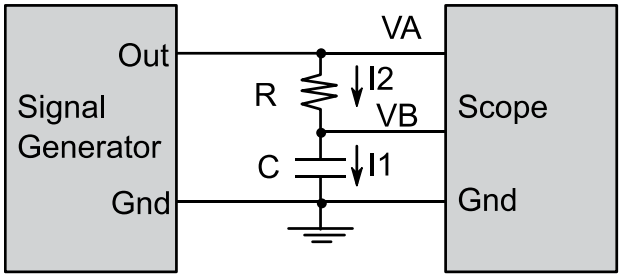


Figure 2. Resistance-capacitor circuit.

Perform AC voltage measurements at four different frequencies. Calculate reactance, X = 1/(2πfC), for frequency.

Calculate

I2 = (VA-VB)/R
I1 = VB/X

Theoretically, the calculation of I2 should equal I1. In summary, fill in the fields of Table 2.

f	R	VA-VB	I2	C	X	VB	I1
100	470			0.47µF			
500	470			0.47µF			
720	470			0.47µF			
1000	470			0.47µF			

Table 2. Experimental verification of capacitor reactance.

*If you are using the **TExaS oscilloscope** make sure the voltages remain within the 0 to 3.3V range. The Texas project will also generate a square wave out of P4.5 that you could use as an AC signal source. You can adjust the square wave frequency.*

Lab 2: Voltage, Current and Power

2.4.3. Voltage-current response of an LED

Begin by reviewing the data sheet of your LED. Choose resistor values that will produce LED currents within the normal operating range of the LED. The 220, 470, 690 ohm values are appropriate for a 10 mA LED. If you have an LED that operates around 1 mA, choose resistors around 1k. You will build the circuit shown in Figure 3.

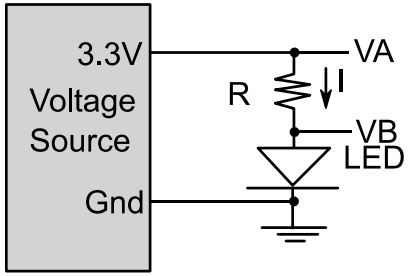


Figure 3. LED test circuit.

Perform LED measurements with four resistance values. E.g., 220 ohms, 470, 690, and 22k ohms. The 690 ohms resistance can be obtained by placing the 220 and 470 resistors in series. For each configuration, calculate the current across the resistor. Calculator power dissipated in the LED.

I = (VA-VB)/R
*P = I*VB*

Try operating the LED connected backwards. You will observe no current flows and the LED is dark. Compare the measured results with the LED data sheet. In summary, fill in the fields of Table 3. Notice the brightness depends on electrical power dissipated in the LED.

R	VA-VB	VB	I	P
220				
470				
690				
22k				

Table3. Experimental measurements in the LED circuit

2.5 Troubleshooting

Measurements don't match theory:

- The most common mistake is the circuit is wired incorrectly.
- Double check you are using the correct resistor and capacitor values.
- This is an experimental lab. Due to the inaccuracies of the measurement devices, your results will not perfectly match theory.

LED doesn't light:

- Excess current will easily damage an LED. If you have damaged your LED, have an instructor check your circuit before applying power to another LED.
- As mentioned earlier, current flows only one way through the LED. The longer lead should have the higher voltage.

2.6 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. The goal of this module is for you to experience voltage, current, and power as seen in resistors, capacitors, and LEDs.

- What is voltage?
- What is current?
- What is power?
- What is Ohm's Law?
- Does Ohm's Law apply to capacitors and LEDs?
- What controls the brightness of the LED?
- What would happen (don't actually do it) if you placed +5V directly across the LED?
- What happens to the power dissipated in a resistor?

2.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. For example,

- For the capacitor circuit, use a dual trace scope to look at (VA-VB) (an indirect measure of capacitor current) and VB (direct measure of capacitor voltage) at the same time. Even though both voltage and current are AC, notice they are not in phase.
- The complex impedance model of a capacitor ($Z=1/(j2\pi fC)$). Notice that $X = |Z|$. The total impedance in Figure 2 is R+Z. The gain of the circuit (VB/VA) can be calculated as Z/(R+Z). Calculate the gain of the circuit at its cutoff frequency.
- Perform the LED experiment on two different LEDs. What is the same? What is different?

2.8 Which modules are next?

We will use the next few labs to overview the processor architecture, and review software development. Module 5 will present the power module, and then we can add modules that will become the robot explorer:

Module 3) Present the processor architecture and develop assembly code.
Module 4) Introduce C and develop some functions needed for the robot.
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

2.9 Things you should have learned

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

- Understand voltage and current in a resistor.
- Understand voltage and current in a capacitor, knowing how frequency affects reactance.
- Understand voltage and current in an LED, knowing the response is extremely nonlinear. Know that electrical power is converted to optical power (brightness).
- Know how to use a voltmeter and oscilloscope.

Module 3

Introduction: ARM Cortex M Architecture

Introduction: ARM Cortex M Architecture

Educational Objectives:

REVIEW Cortex M architecture
UNDERSTAND registers, memory, assembly instructions
DEVELOP logic and arithmetic functions in assembly
LEARN how functions work, and where data is stored
DESIGN, BUILD & TEST A COMPONENT
Nonlinear conversion function for an IR distance sensor

Prerequisites (Module 1)
- Running code on the LaunchPad using CCS (Module 1)

Recommended reading materials for students:
- Volume 1 Section 1.7, Chapter 3, and Section 5.3
  Embedded Systems: Introduction to the MSP432 Microcontroller
  ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or

- Volume 2 Sections 1.1, 2.1, and 2.5
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

In this class, we will use the TI Launchpad Development Kit with the MSP432 microcontroller, which includes a Cortex-M processor and a suite of input/output devices derived from the MSP430 family of low power microcontrollers. **Architecture** is the manner with which the processor, random access memory (RAM), read only memory (ROM), and input/output (I/O) ports are combined to create the microcontroller. See Figure 1.

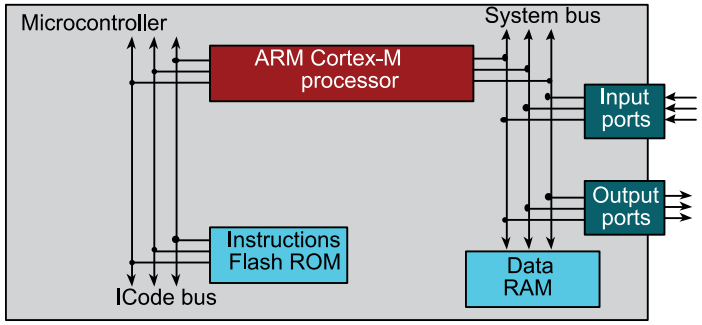


Figure 1. Architecture of an ARM Cortex M microcontroller.

This module serves as a brief introduction to the Cortex M microcontroller. Even though we typically program embedded systems in C, it makes sense to understand a little bit how the microcontroller executes software. Understanding some of these low level details will make it easier to make high level software design decisions. Examples where knowing low-level details make for better high-level decisions include: local verses global variables, numbers verses pointers, numerical overflow during calculations, numerical dropout during divide and right shift operations, integer versus floating point calculations, and interrupts.

There are two reasons we must learn the assembly language of the computer with which we are using. Sometimes, but not often, we wish to optimize our application for maximum execution speed or minimum memory size, and thus writing pieces of our code in assembly language is one approach to such optimizations. The most important reason, however, is that by observing the assembly code generated by the compiler for our C code we can truly understand what our software is doing. Based on this understanding, we can evaluate, debug, and optimize our system. So the goal of this module is not for you to become proficient in assembly language, but rather to learn enough so you can interpret the assembly code generated by the C compiler.

An **assembler** is system software that converts low-level assembly language program (human readable format) into object code (machine readable format). Typically, one line of assembly language creates one machine instruction, and this translation is simple and obvious. Writing in assembly exposes the low-level details of the architecture.

A **linker** builds a single software system by connecting (linking) software components. In CCS, the **build** command performs both assembly and linking. In an embedded system, the **loader** will program object code into flash ROM. We place object code in ROM because ROM retains its information if power is removed and restored. In CCS, the **Debug** command performs a load operation and starts the debugger.

A **debugger** is a set of hardware and software tools we use to verify system is operating correctly. The two important aspects of a good debugger are control and observability.

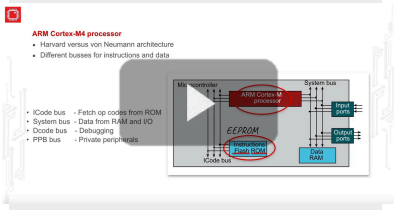
In the lab associated with this module, you will develop and test an assembly function typical of one the robot might use to perform a numerical calculation. In particular, the function will convert ADC measurements from a sensor into distance to the wall. In developing and debugging this function, you will gather important insights on how the Cortex-M processor executes software.

3. TI-RSLK Module 3 – ARM Cortex M

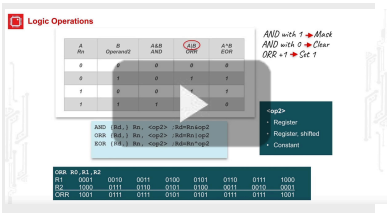
This module serves as a brief introduction to the ARM Cortex-M microcontroller, assembly programming language and some debugging techniques. Understanding how the processor works is essential for the design of embedded systems, such as the one used in your robot.

Optionally, download all the curriculum documents for Module 3.

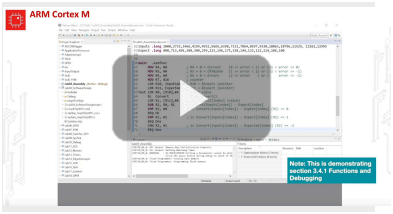
3.1 **TI-RSLK Module 3 - Lecture video part I - ARM Cortex M Architecture**
Understanding how the processor works is essential for the design of embedded systems, such as the one used in your robot.



3.2 **TI-RSLK Module 3 - Lecture video part II - ARM Cortex M Assembly**
In this module you will develop and test an assembly function the maze robot might use to perform a numerical calculation.



3.3 **TI-RSLK Module 3 - Lab video 3.1 - Debugging the solution, visualization, breakpoint and step**
The purpose of this lab is to introduce the architecture of the Cortex M.



Module 3

Lab 3: ARM Cortex M Architecture

Lab 3: ARM Cortex M Architecture

3.0 Objectives

The purpose of this lab is to introduce the architecture of the Cortex M.

1. You will learn about registers, RAM, and flash ROM.
2. You will write an assembly function with input and output parameters, which includes conditional and arithmetic operations.
3. You will learn debugging techniques like single stepping, breakpoints, and watch windows.
4. You will use an automated test approach called black-box functional testing to verify your algorithm is operating properly

Good to Know: We will be programming the robot challenge in C. However, the compiler converts the C code into assembly code. It is this low-level code that actually runs on the MSP432, which is a Cortex-M microcontroller. In this lab, you will experience some of the details of how the microcontroller executes software. Knowing these low-level details will make you a better high-level software developer.

3.1 Getting Started

3.1.1 Software Starter Projects

Look at these three projects:
SimpleProject_asm (a simple project that implements a random number generator),
LinearInterpolation_asm (an implementation of sine), and
Lab03_Assembly (starter project for this lab)

3.1.2 Student Resources (in datasheets directory-Links)

spmu159a.pdf, Cortex-M3/M4F Instruction Set

3.1.3 Reading Materials

Volume 1 Section 1.7, Chapter 3, and Section 5.3
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 1.1, 2.1, and 2.5
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller",

3.1.4 Components needed for this lab

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

3.1.5 Lab equipment needed (none)

3.2 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 better understand how the computer performs tasks. We expect most students will complete the robot challenge programming in C. However, in this lab you will write a simple function in assembly.

Note: In the robot challenge 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 which will be placed on the robot to measure distance.

In lab of module 4 you will develop a C function that converts raw ADC samples into a distance for the GP2Y0A21YK0F proximity sensor.

Let **n** be a 14-bit sample from the ADC (0 to 16383), 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 maximum measurement distance for the sensor is 800 mm, so if the ADC value is less than 2552, your function should return 800. The C prototype for your function is

int32_t Convert(int32_t n);

However, since you are writing the function in assembly, you must adhere to a programming standard, called **ARM Architecture Procedure Call Standard** (AAPCS). There are many components of this standard, but the ones relevant to this lab include:

- If there is one input parameter, it is passed in R0
- If there are two input parameters, they are passed in R0, R1
- If there are three input parameters, they are passed in R0-R2
- If there are four input parameters, they are passed in R0-R3
- If there is an output parameter, it is returned in R0
- The function can modify R0-R3, R12 freely
- If a function wishes to use R4-R11 then it must save and restore them using the stack.
- If a function calls another, then it must save and restore LR
- Functions must balance the stack

Adhering this standard will allow you to develop assembly code that can be called from C, and allow your C code to be called from assembly. In particular, the compiler will adhere to this standard when creating object code.

3.3 Experiment set-up

This lab uses the LaunchPad without any input/output hardware.

3.4 System Development Plan

3.4.1 Functions and debugging

In this lab section you will build and debug the **SimpleProject_asm** example. Using the debugger, observe the input and output parameters of the function while you single step through the main program.

Answer the following:

i) How are data passed into **Seed**?
ii) How are results passed back from **Rand**?
iii) What happens to the LR register when a function is called?
iv) How does a function return?
v) How does the software access global RAM?
vi) What is the difference between storing data in a register and storing it in global RAM?
vii) Where is the machine code stored?
viii) What do **.data** and **.text** mean?
ix) Where are the constants 1664525 and 1013904223 stored?
x) You can observe the variables M and n by placing their addresses into a **Memory Browser** window.
xi) Using the step-over command, execute the Rand function multiple times and observe the values in M and n. In particular, look at bit 0 of M; what pattern do you see in bit 0?

Next you will build and debug the **LinearInterpolation_asm** project. If you are unfamiliar with “linear interpolation”, do an internet search on the topic to better understand the math used in this project.

Using the debugger, place a breakpoint inside the **Sin** function, and use the debugger observe the values of the **registers** during one execution of the **Sin** function. From a programming theory standpoint, these registers are considered **local variables** for the function.

Answer the following:

i) Can you prove the three subtract instructions will never overflow, when calculating (Ix-x1), (y2-y1), and (x2-x1)?
ii) Can you prove the multiply instruction will never overflow, when calculating (y2-y1)*(Ix-x1)?
iii) Can you prove it will never divide by zero?
iv) Why do we use SDIV instead of UDIV for this function?

Observe how this main program tests the **Sin** function. We call the form of testing in main.asm **Black Box** functional testing, because the testing just sets inputs and observes outputs. In other words, we look at the outside of the software, and not probe any of the internal details of the function. Black box testing looks at the overall functionality of what software does without know of how it works.

3.4.2 Distance Conversion

Write an assembly function that converts raw 14-bit ADC data to distance in mm. Use **.field** statements to encapsulate the calibration parameters.

```
IRSlope  .field 1195172,32
IROffset .field -1058,32
IRMax    .field 2552,32
```

Lab 3: ARM Cortex M Architecture

You can use the **main** program delivered as part of the **Lab_Assembly** project to test your **Convert** function. Similar to **LinearInterpolation_asm**, this testing approach is **Black Box functional testing**. This test program contains 16 test cases (inputs and expected outputs). The expected results are plotted as Figure 1.

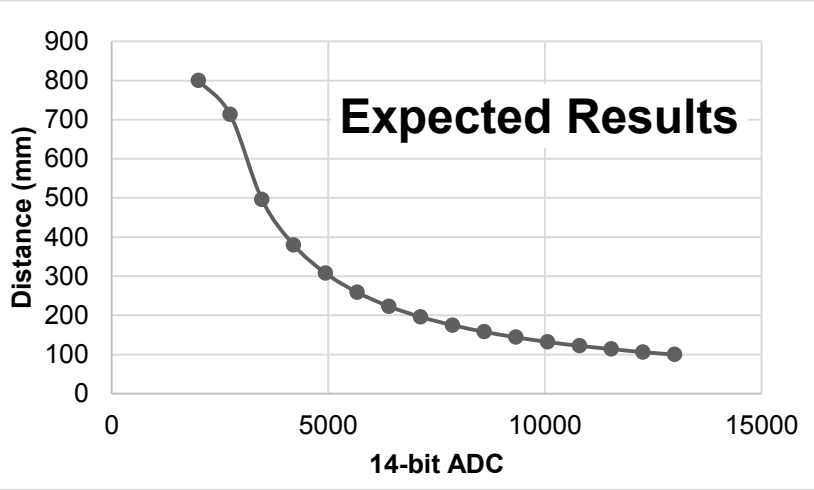


Figure 1. Expected results for the GP2Y0A21YK0F conversion function.

*Run **main** and compare your results with expected values. It is ok if your results differ by ±1 (which could be due to rounding).*

3.4.3. Observing Compiler-Generated Assembly Code

Revisit one of the C examples you ran as part of Lab 1. Within the debugger, open a Disassembly window. Single step the C code and observe the actual instructions

3.5 Troubleshooting

Convert doesn't work:

- Using **main**, find an input value that does not work, write a simple main program that calls your function with just that input, and single step your program comparing your internal calculations with expected values. Observing internal values is called **white-box** testing.
- 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 testing procedure.

3.6 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. The goal of this module is for you to know enough assembly language to be able to interpret the machine-executable code generated by the compiler.

- What information do we store in ROM? Why?
- What information do we store in RAM? Why?
- What information do we store in R0-R12 registers? Why?
- How are R4-R11 different from R0-R3, R12?
- How is the LR used?
- How is the SP used?
- How is the PC used?
- How do functions work? Input parameters? Return parameter?
- Can you prove the (**n** – 1058) subtraction never overflows?
- Can you prove the division never attempts a divide by zero?
- Using integer division, what is the result of 1/**n** for any values of **n** greater than 1? This error (loss of information) is called **dropout**.
- The input is a 14-bit number (0 to 16383), but the output is only a 10-bit number (0 to 800). This reduction of four bits is a mild form of **dropout**. How could you have reformulated the problem to have less dropout?
- Notice that **SimpleProject_asm** project uses just one source file, while **LinearInterpolation_asm Lab_Assembly** projects use two source files. How are these two files used? What is the advantage of separating the implementation software from the testing software?
- List the debugging techniques used in this lab.

3.7 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 that tries every possible 14-bit input from 0 to 16383. How would you generate the test cases? How would you change the main program? What are the advantages of exhaustive testing?
- The robot can have multiple proximity sensors. Redesign the **Convert** function to handle three sensors, where each sensor has a unique set of three calibration coefficients (IRSlope IROffset IRMax).
- Use the debugger to estimate the time it takes to execute your **Convert** function.
- The Cortex M supports floating point arithmetic. Implement a floating point version of the function and develop a means to test it. Compare the accuracy and execution times for the two versions.

3.8 Which modules are next?

We will use the next few labs to create 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 4) Introduce C and develop some functions needed for the robot.
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.

3.9 Things you should have learned

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

- Understand how the processor uses registers during execution
- Discover the differences between RAM and ROM and how the software uses each.
- Perform arithmetic calculations in assembly with addition, subtraction, multiplication, and division
- Understand how constants are stored on the microcontroller
- Make decisions with conditional branch assembly statements
- Use the debugger to single step and visualize variables
- Perform functional testing

Module 4

Introduction: Software Design Using MSP432

Introduction: Software Design Using MSP432

Educational Objectives:

REVIEW C programming
UNDERSTAND conditional statements and loops in C
DEVELOP logic and arithmetic functions
LEARN how to debug simple programs in C
DESIGN, BUILD & TEST A SOFTWARE COMPONENT

As the robot explores its world, it must make decisions. In the lab associated with this module, you will write software that takes input from three distance sensors and determines if one of eight possible scenarios is present, see Figure 1. The actual sensors will be interfaced in Lab 15, but in this lab you will write software to be used in the robot later.

Prerequisites (Module 1)

- Running code on the LaunchPad using CCS (Module 1)

Recommended reading materials for students:

- Volume 1 Chapter 1, Sections 2.8, 5.1, 5.2, 5.3, 5.6, and 5.8
  Embedded Systems: Introduction to the MSP432 Microcontroller ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or

- Volume 2 Sections 1.4, 1.5, 3.1, 3.2, 3.3, and 3.4
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

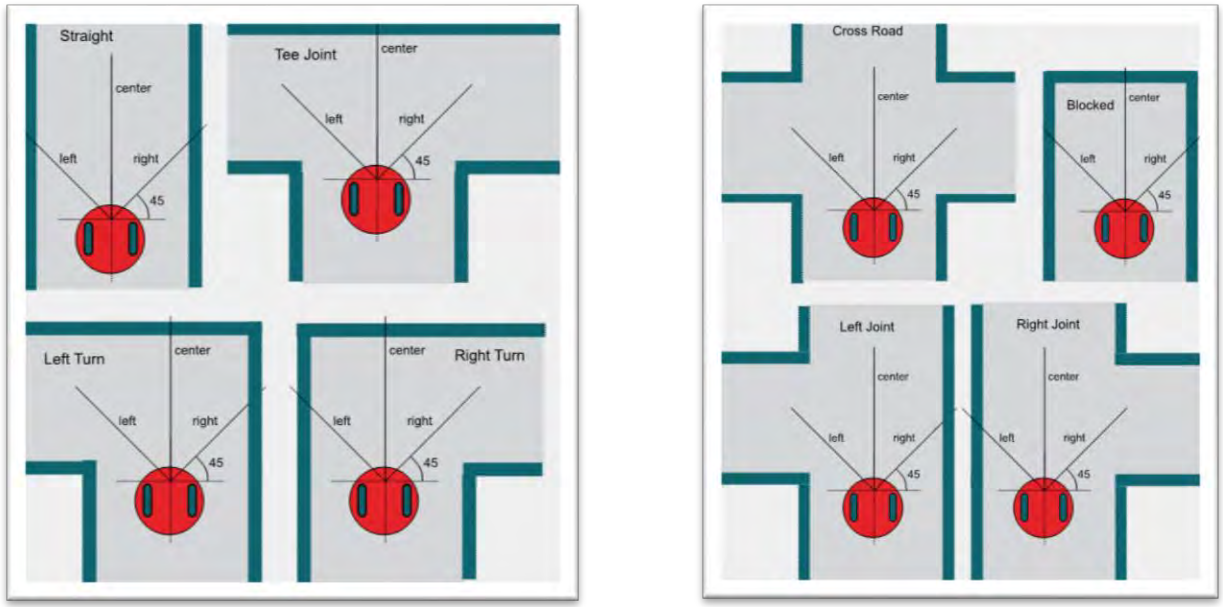


Figure 1. Eight possible scenarios as the robot explores the maze. As the robot approaches an intersection, it first determines what alternative paths exist, and then it chooses which way to go

Introduction: Software Design Using MSP432

This module serves as a brief introduction to C. C is a general-purpose programming language initially developed by Dennis Ritchie between 1969 and 1973 while at AT&T Bell Labs. In 1999, a professional standard version of C, called C99, was defined. In this class we will write our software in C99, because it is prevalent in industry.

A **compiler** is system software that converts a high-level language program (human readable format) into object code (machine readable format). It produces software that is fast but to change the software we need to edit the source code and recompile.

The Project Explorer in CCS shows us the various components used for each project. A **linker** builds a single software system by connecting (linking) software components. In CCS, the **build** command performs both a compilation and a linking.
In an embedded system, the **loader** will program object code into flash ROM. We place object code in ROM because ROM is retains its information if power is removed and restored. In CCS, the **Debug** command performs a load operation and starts the debugger.

A **debugger** is a set of hardware and software tools we use to verify system is operating correctly. The two important aspects of a good debugger are control and observability.

Before we write software, we need to develop a plan. Software development is an iterative process. Even though we list steps the development process in a 1,2,3,4 order, in reality we cycle through these steps over and over. I like to begin with step 4), deciding how I will test it even before I decide what it does.

1) We begin with a list of the inputs and outputs. This usually defines what the overall system will do. We specify the range of values and their significance.
2) Next, we make a list of the required data. We must decide how the data is structured, what does it mean, how it is collected, and how it can be changed.
3) Next we develop the software algorithm, which is a sequence of operations we wish to execute. There are many approaches to describing the plan. Experienced programmers can develop the algorithm directly in C language. On the other hand, most of us need an abstractive method to document the desired sequence of actions. Flowcharts and pseudo code are two common descriptive formats. There are no formal rules regarding pseudo code, rather it is a shorthand for describing what to do and when to do it. We can place our pseudo code as documentation into the comment fields of our program. Next we write software to implement the algorithm as define in the flowchart and pseudo code.
4) The last stage is debugging. Learning debugging skills will greatly improve the quality of your software and the efficiency at which you can develop code.

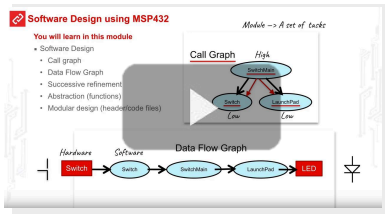
In the lab associated with this module, you will develop and test some software functions that will be used later in the explorer robot. In particular, the first function will convert ADC measurements from a sensor into distance to the wall, and the second function will take three distance measurements and classify the situation into the most likely scenario.

4. TI-RSLK Module 4 – Software Design using MSP432

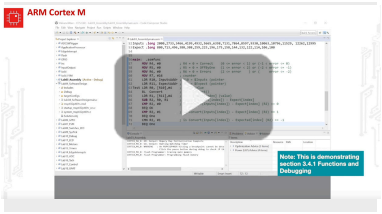
This module is an introduction to C, a general-purpose programming language, in addition to the concepts of compiling and debugging using the MSP432 and TI Code Composer Studio™. Debugging skills are a valuable tool when developing complex systems involved with robotics.

Optionally, download all the curriculum documents for Module 4.

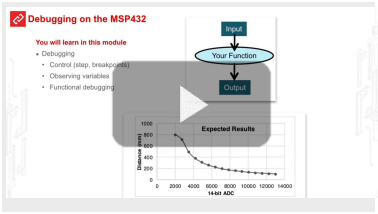
4.1 **TI-RSLK Module 4 - Lecture video part I - Software design using MSP432 - Design**
Learn software design through a call graph, data flow graph, successive refinement, abstraction (functions) and modular design (header/code files).



4.2 **TI-RSLK Module 4 - Lecture video part II - Software design using MSP432 - C programming**
In this module, you will develop and test software functions that will be used in the maze robot.



4.3 **TI-RSLK Module 4 - Lecture video part III - Software design using MSP432 - Debugging**
Learn debugging on the MSP432 through controls (step, breakpoints), observing variables and functional debugging.



4.4 **TI-RSLK Module 4 - Lab 4 video 4.1 - Debugging the solution, visualization, variables, step over**
The purpose of this lab is to interface a line sensor that the robot will use to explore its world.

4.5 **TI-RSLK Module 4 - Lab video 4.2 - Debugging the solution, visualization, breakpoint, step over**
The purpose of this lab is to introduce software design.

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

Volume 1 Chapter 1, Sections 2.8, 5.1, 5.2, 5.3, 5.6, and 5.8
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 1.4, 1.5, 3.1, 3.2, 3.3, and 3.4
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

4.1.4 Components needed for this lab

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.

However, since you are writing the function in assembly, you must adhere to a programming standard, called **ARM Architecture Procedure Call Standard** (AAPCS). There are many components of this standard, but the ones relevant to this lab include:

- If there is one input parameter, it is passed in R0
- If there are two input parameters, they are passed in R0, R1
- If there are three input parameters, they are passed in R0-R2
- If there are four input parameters, they are passed in R0-R3
- If there is an output parameter, it is returned in R0
- The function can modify R0-R3, R12 freely
- If a function wishes to use R4-R11 then it must save and restore them using the stack.
- If a function calls another, then it must save and restore LR
- Functions must balance the stack

Adhering this standard will allow you to develop assembly code that can be called from C, and allow your C code to be called from assembly. In particular, the compiler will adhere to this standard when creating object code.

Lab 4: Software Design Using MSP432

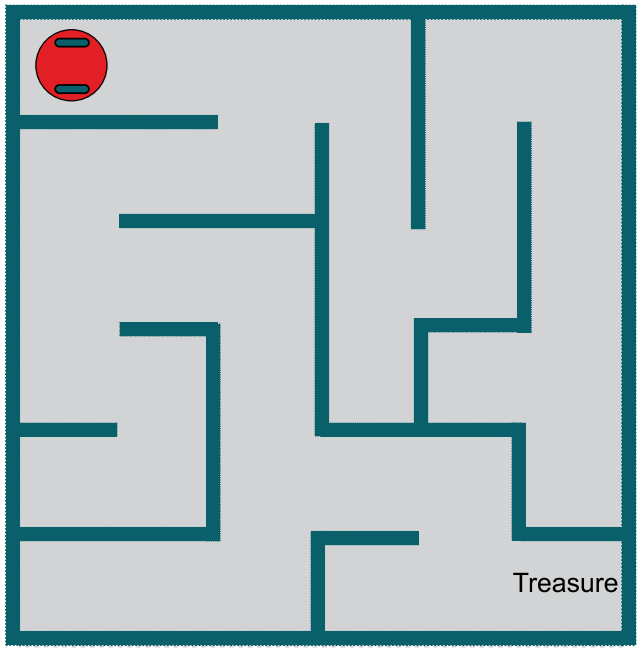


Figure 1. Possible robot challenge of exploring the world.

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), will be needed by the robot, 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 allow to 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

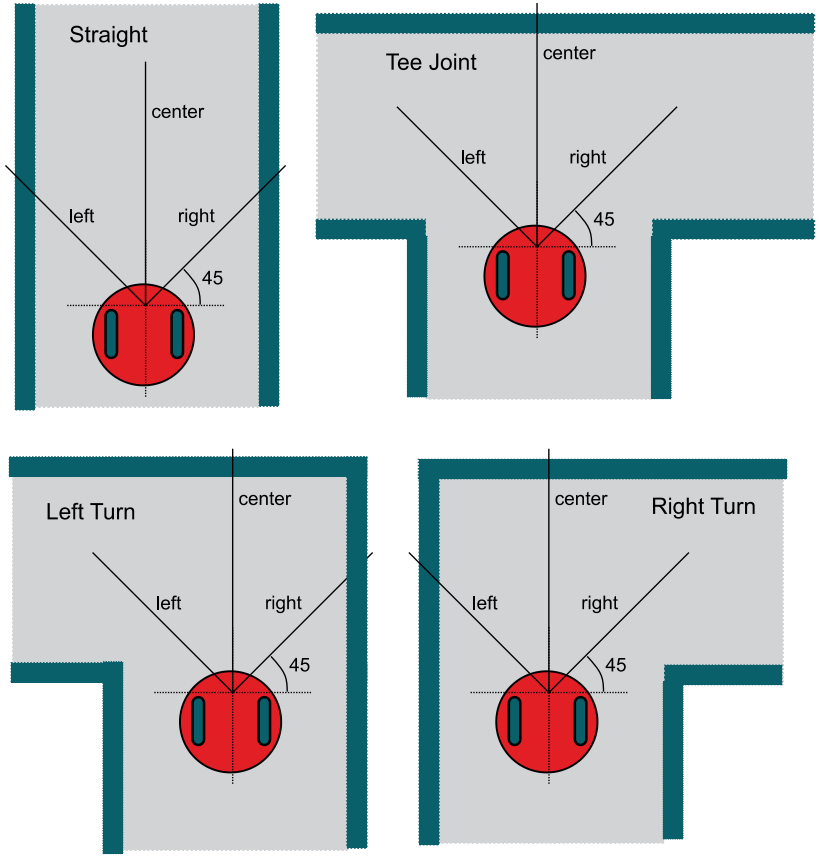


Figure 2. Four possible scenerios as the robot approaches a decision point.The three variables (left, center, and right) are defined as the distance from the center of the robot to the wall.

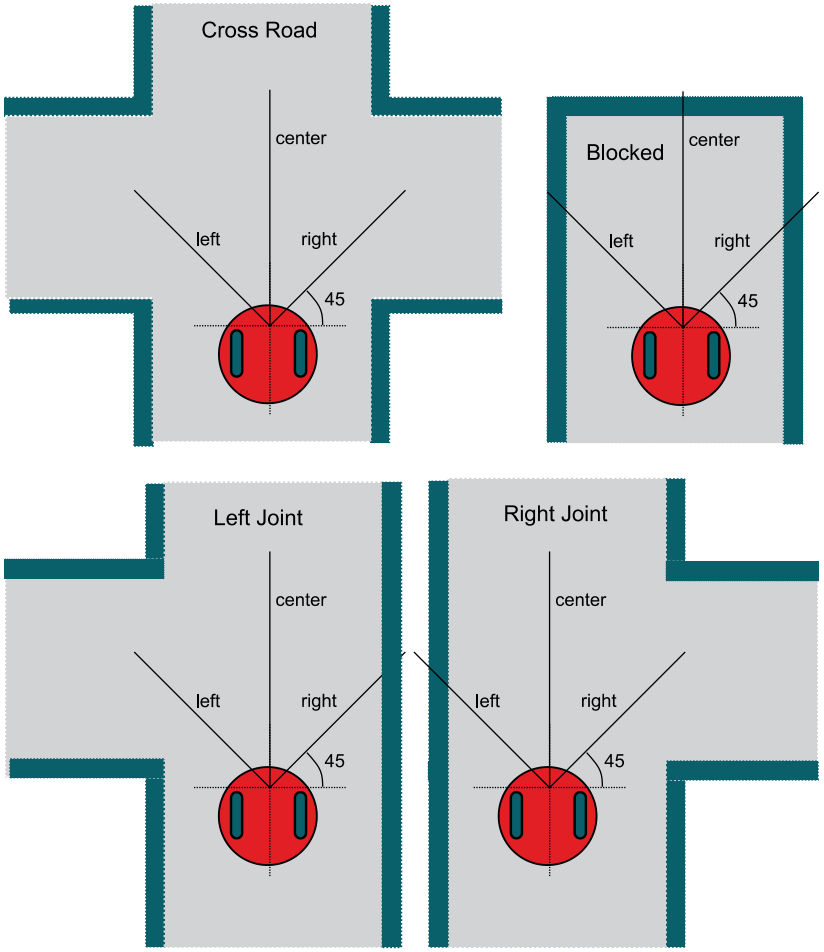


Figure 3. Four more scenerios as the robot approaches a decision point.

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.

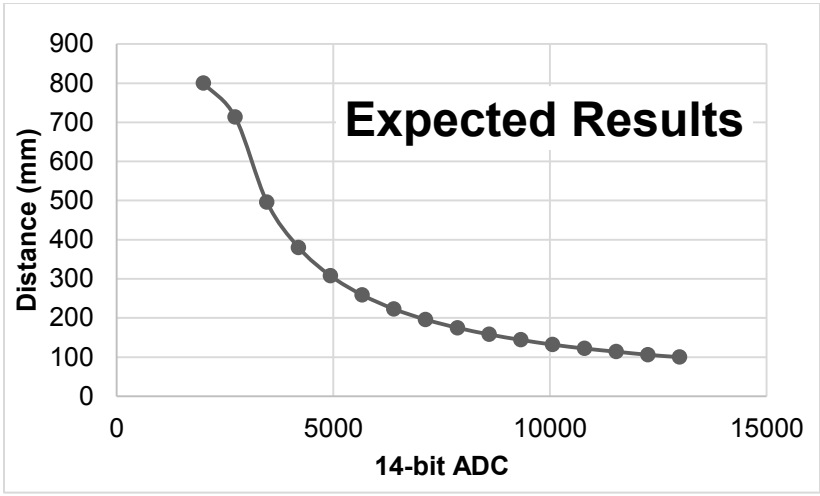


Figure 4. Expected results for the GP2Y0A21YK0F conversion function.

```
// 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 instructors ***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 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 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 with Program 4_3, which you can find in the starter project. 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

Module 5

Introduction: Battery and Voltage Regulation

Introduction: Battery and Voltage Regulation

Educational Objectives:

MEASURE Voltage, current, and energy for a battery
UNDERSTAND Voltage regulation for the robot
INTERFACE The circuits needed to power the robot from batteries

Prerequisites (Module 2)

- Voltage, current, energy, power (Module 2)
- Resistance, capacitance (Module 2)

Recommended reading materials for students:

- Volume 1 Section 1.1
  Embedded Systems: Introduction to the MSP432 Microcontroller, ISBN: 978-1512185676, Jonathan Valvano, copyright © 2017

  or

- Volume 2 Section 9.2
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright © 2017

Every embedded system needs power to operate. The source of power could be
- 120 VAC 60 Hz, with an AC to DC converter
- DC power, like +5V on USB or +12V in an automobile
- Battery
- Energy harvesting like solar or EM field pickup

When debugging the LaunchPad, you use +5V from the PC via the USB cable. However, to run the robot autonomously, it will need battery power. The battery voltage is not constant; it decreases with age and use. Therefore, you will use a **regulator** to provide a constant voltage to power most of the electronics for the robot. In this module, we will introduce two types of regulators: linear and switching. There are many considerations when choosing a regulator, and we will discuss some of these considerations.

You will power the robot motors directly off the battery voltage. The Romi Chassis (Pololu part #3502) can hold 6 AA batteries. If you use NiMH batteries (1.2V each), this will create a +7.2V source for the robot. The motors do not need a constant voltage to operate, and running directly off the batteries is the most efficient use of energy. As you might imagine, the motors use most of the power required by the robot.
The robot will take the battery +7.2V input and create a +5V regulated power source. In particular, you will use the Motor Driver and Power Distribution Board for Romi Chassis (Pololu part #3543). We will explain the battery and voltage regulation in this module. You will connect the +5V regulated power source to the LaunchPad, and the LaunchPad will create a +3.3V power source using its own regulator. The LaunchPad powers the MSP432 with this +3.3V. The MSP432 itself has regulators inside the chip. For example, V_{CORE} is the internal voltage at which the processor operates, and it is typically +1.2V. You will power the motors directly off the battery, some of the external devices with +5V and others with +3.3V.
The **energy** (E in joules) stored in a battery can be calculated from voltage (V in volts), current (I in amps), and time (t in seconds). Energy has neither polarity nor direction. The energy rating for the battery is given in amp-hr, because the assumption is the voltage is constant. The NiMH batteries listed in the lab bill of materials (BOM) are rated at 1900 mA-hr. This means the battery can supply 1 amp for 1.9 hours. Six of these batteries, placed in series, can supply 7.2V at 1 amp for 1.9 hours.
In the lab associated with this module, see Figure 1, you will study the batteries, measuring their energy storage. Next, you will build/interface the circuits needed to power the LaunchPad off batteries.

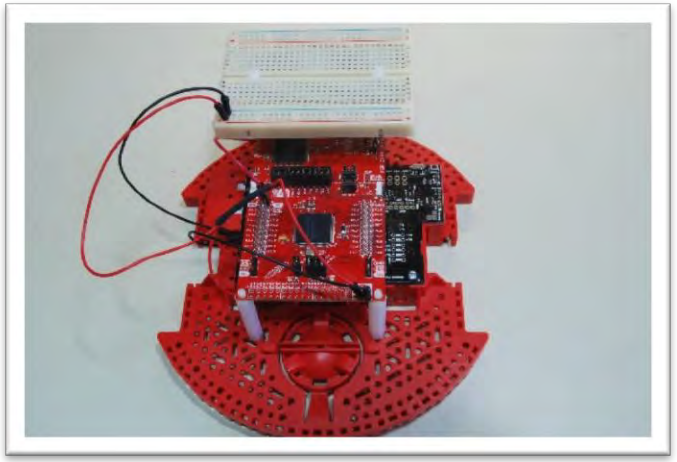


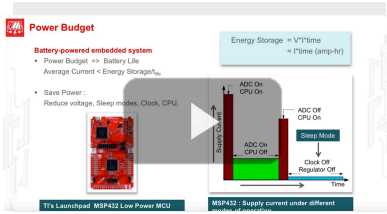
Figure 1. Romi Chassis, Motor Driver and Power Distribution Board, LaunchPad, and small protoboard.

5. TI-RSLK Module 5 – Battery and voltage regulation

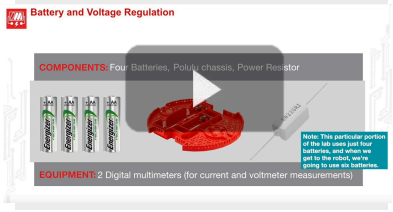
The purpose of this module is to learn how to power your robot. To run the robot (motor and other systems) you will need batteries and a regulator to provide constant voltage. Understanding the relationship between voltage current and power is an essential component of robot system design.

Optionally, download all the curriculum documents for Module 5.

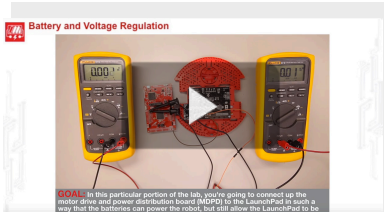
5.1 TI-RSLK Module 5 - Lecture video part I - Battery and voltage regulation
Learn battery power sources, voltage regulation (constant voltage) and performance measurements.



5.2 TI-RSLK Module 5 - Lab video 5.1 - Measure voltage and current from battery
The purpose of this lab is to study the batteries and how they are used to power the robot.



5.3 TI-RSLK Module 5 - Lab video 5.2 - Connecting motor driver and power distribution board
The purpose of this lab is to power the system from the batteries.



Module 5

Lab 5: Battery and Voltage Regulation

Lab 5: Battery and Voltage Regulation

5.0 Objectives

The purpose of this lab is to build the electronics needed to power the robot. During debugging, +5V power will be available from the PC via the USB cable. However, during autonomous running, the robot will derive power from batteries. In this module,

1. You will learn about voltage, current, and power for the battery.
2. You will perform experiments with the battery.
3. You will configure the driver board needed for the robot.
4. You will run the MSP432 LaunchPad under battery power.

Good to Know: **Power management is an important aspect of embedded systems. Many considerations affect system performance. These considerations include, but are not limited to, voltage, current, battery life, size, weight, and power-line ripple (noise).**

5.1 Getting Started

5.1.1 Software Starter Projects (none)

Note: Please do not use the voltmeter, oscilloscope or logic analyzer created by TExaS for this lab. Voltages applied to inputs of the MSP432 must remain between 0 and 3.3V. Voltages outside this range will damage the MSP432.

5.1.2 Student Resources

Yageo LR_SQP NSP_2013.pdf	Data sheet for 10W resistor
MotorDriverPowerDistribution.pdf	Data sheet for power board
Keystone_4-40-NylonStandoff.pdf	Holds up LaunchPad
Keystone_4-40-Screw.pdf	Attach standoff to Romi Chassis
Keystone_4-40-Nut.pdf	Holds LaunchPad
Keystone_4-40-Standoff.pdf	Holds breadboard
Pololu Romi Chassis User's Guide.pdf	How to build the robot

5.1.3 Reading Materials

Volume 1 Section 1.1
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Section 9.2
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

Read the specifications on the Motor Driver and Power Distribution board website https://www.pololu.com/product/3543, https://www.pololu.com/docs/0J68

5.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	Romi Chassis Kit - Red	Pololu	3502
1	Motor Driver and Power Distribution Board for Romi	Pololu	3543
1	Rechargeable Battery, Pack of 4, Metal Hydride 1300 mAh, 1.2V, AA	Energizer	626831
1	Wirewound 5W, 5%, 10 ohm	WELWYN	WHS5-10RJT075
4	0.5 in 4-40 machine screw	Pololu	1962
4	1.375in 4-40 Nylon standoff	Keystone	4809
2	0.187in 4-40 metal nut	Keystone	4694
2	0.75in 4-40 metal standoff	Keystone	2204
2	0.5in 4-40 Nylon machine screw	Keystone	9529

Lab 5: Battery and Voltage Regulation

5.1.5 Lab equipment needed

Oscilloscope (one or two channels at least 10 kHz sampling)
Voltmeter, current meter

5.2 System Design Requirements

The two goals of this lab are to study the behavior of batteries and configure the **Motor Driver and Power Distribution Board (MDPDB)** for use in the robot. The ultimate goal is to create a regulated +5V power source for the robot. The LaunchPad should require less than 100 mA at +5V.

The robot needs a mechanism to securely hold the batteries during operation. The **Romi Chassis (https://www.pololu.com/docs/0J68)** has mechanical support for six AA batteries. In this lab, we assume you have six 1.2V NiMH batteries, creating +7.2V for the robot. Fully charged NiMH batteries will create about 8.2V. Refer to the data sheet for the power board for other configurations.

We define the **energy storage** of a battery in amp-hours, because the voltage is assumed constant. The standard units of energy are watt-seconds (1 W-sec is 1 J). One can estimate the operation time of a battery-powered embedded system by dividing the energy storage by the average current required to run the system. The **power budget** embodies this concept. Let E be the energy storage specification of the battery in mA-hour and t_{life} be the desired lifetime of the product (in hours); then we can estimate the average current our system is allowed to draw (in mA):

$$\text{Average Current} \leq E / t_{life}$$

5.3 Experiment set-up

In order to test the robot's batteries, we begin by constructing the battery storage components for the Romi Chassis. Figure 1 shows the bottom of the chassis with the six NiMH batteries. Without the **MDPDB** attached, there should be about 4*1.2V = 4.8V across BAT1+ to BAT1-, and there should be about 2*1.2V=2.4V across BAT2+ to BAT2-. The **MDPDB** will connect BAT1- to BAT2+, and it will connect BAT2- to ground, so there will be 7.2V from BAT1+ to ground.

You should perform the experiment described in 5.4.1 before soldering the **MDPDB** to the battery terminals.

Note: For safety reasons we recommend testing the battery at currents less than ½ amp. Please also make sure the power to the resistor is within limits.

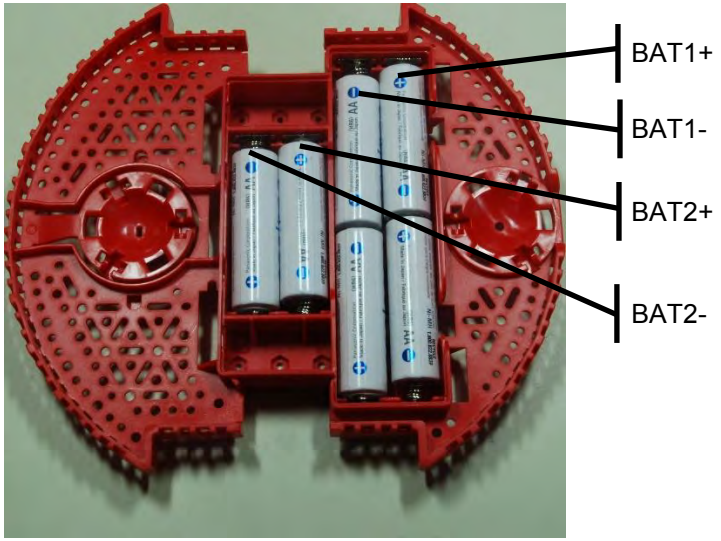


Figure 1. Romi Chassis holds 6 AA batteries.

The first step is to study the behavior of your battery using a simple set up as shown in Figure 2. It is important to limit the power to the resistor to less than its power rating. The power delivered to a resistor is

$$P = I*V = V^2/R = I^2*R$$

The resistor listed in the BOM (components needed) has a 10W rating. One possible configuration uses the four NiMH batteries (BAT1+ to BAT1-) to create a +4.8 V battery source. In this experiment we will use the 10 Ω resistor as a fixed load. The current draw will be 4.8V/10Ω = 480 mA. The power will be $4.8^2/10$ = 2.3W. At this power, the resistor will get hot, but it will operate below than the 10W max power rating. Because the battery storage is 1900 mA-hr, it will take 1900/480=4 hours to perform the discharge experiment.

Lab 5: Battery and Voltage Regulation

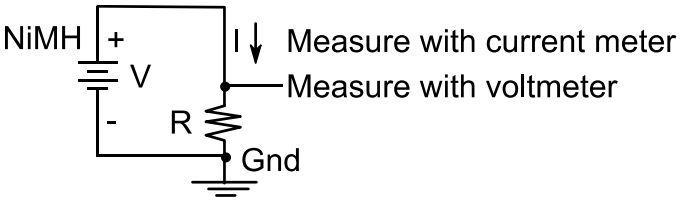


Figure 2. Battery circuit.

The second step is to configure the **MDPDB** for use on the robot. Figure 3 shows the locations of the power-related pins on the motor driver board.

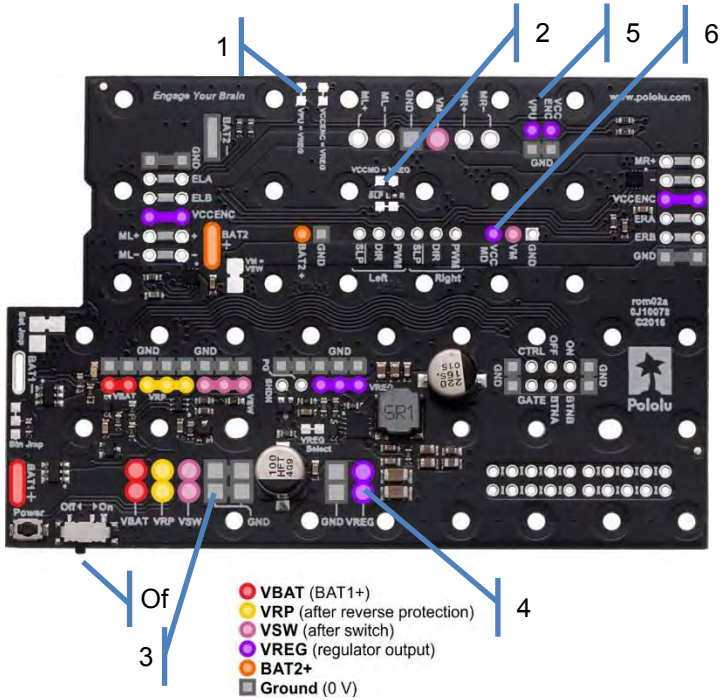


Figure 3. Motor Driver and Power Distribution Board for Romi Chassis

(source : Pololu.com)

Follow these steps to configure the motor driver board for the robot:

1) Cut the VPU—VREG jumper on **the MDPDB**
2) Cut the VCCMD—VREG jumper on the **MDPDB**
3) Solder a ground wire from the **MDPDB** to ground on the LaunchPad
   We suggest you review the entire MDPDB User's Guide
   Make sure not to use holes needed later for the motors
4) Connect VREG (+5V) from the **MDPDB** to +5V on the LaunchPad
   You will need a way to connect/disconnect +5V.
   We suggest you solder one end of a wire to the VREG signal on the **MDPDB** and use a female header to connect it to +5V on the LaunchPad.
5) Connect VPU from the **MDPDB** to 3.3V on the LaunchPad
6) Connect VCCMD from the **MDPDB** to 3.3V on the LaunchPad
7) Solder a wire with a male header to the +3.3V power. Solder a second wire with a male header to ground. These two will be used to bring power to the solderless breadboard (shown in Figure 4).

Figure 4 shows the partially completed power system.

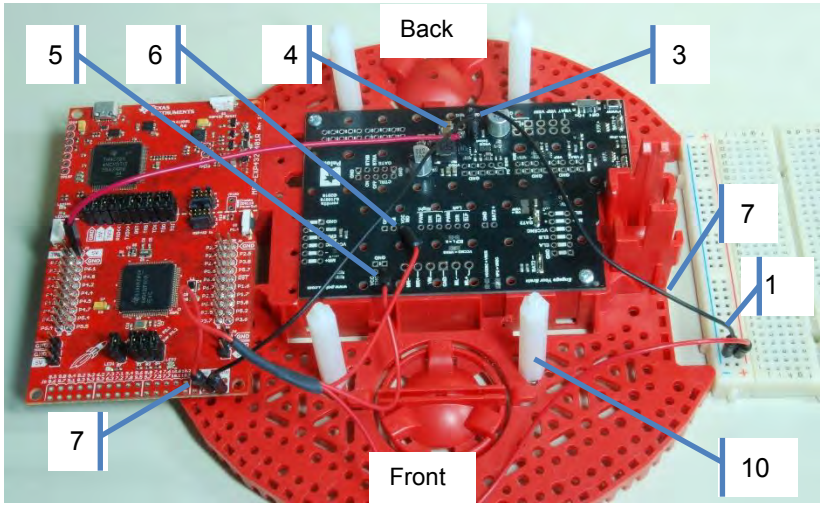


Figure 4. Motor Driver and Power Distribution Board connected to the LaunchPad. Grounds connected. VREG (MDPDB) to +5V (LaunchPad). +3.3V (LaunchPad) to VPU (MDPDB) and VCCMD (MDPDB).

Lab 5: Battery and Voltage Regulation

Follow these steps to mechanically attach the boards to the chassis:

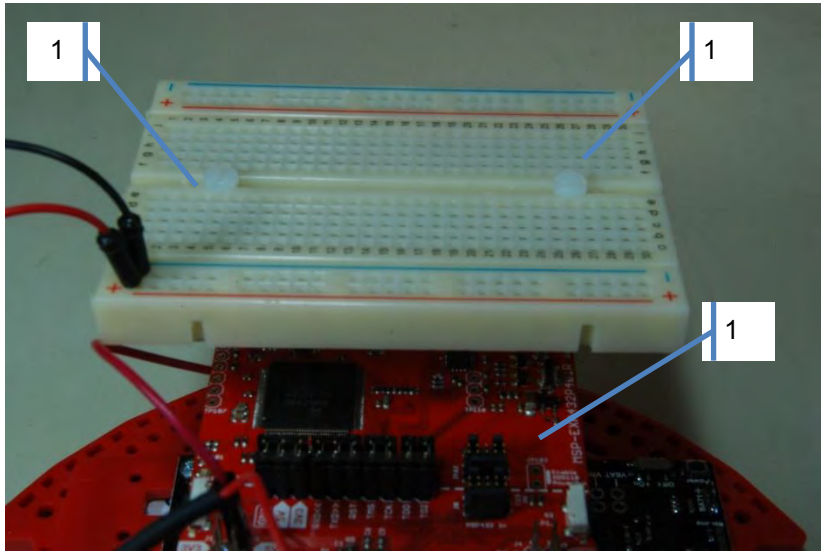


Figure 5. Solderless breadboard attached to the robot.

8) Drop the four battery terminals from the top, looking at Figure 1 and orienting the Bat+ signals with the positive side of the battery (tab) and the Bat- signals with the negative side of the battery (spring). Screw the **MDPDB** to the chassis using the two screws and nuts provided in the **MDPDB** kit. See Figure 1 and follow directions from Pololu.
9) Double check the positive and negative alignment. Solder the four battery terminals from the chassis to the **MDPDB**. We suggest you solder the terminals after it is aligned with the screws and nuts. The board can still be detached from the robot by removing the screws and squeezing the battery springs. See Figure 1 and follow directions from Pololu.
10) Attach four 1.375in 4-40 Nylon standoffs to the robot chassis using 0.25in screws. Align the standoffs with the four holes on the LaunchPad. Depending on where you choose to mount the LaunchPad, you may have to increase the side of the hole on the chassis to ¼ inch to accept the 4-40 screw.
11) Place the LaunchPad on top of the standoffs orienting the USB cord out the back of the robot (the LaunchPad shown in Figure 4 will not be rotated before placement). Use two nuts in the front and two 0.75in 4-40 metal standoffs in the back to secure the LaunchPad.
12) Drill two ¼ inch holes in the solderless breadboard to align with the two 0.75 inch standoffs. Attach the breadboard to the standoffs using two ½ inch Nylon screws. See Figure 5.
13) Attach the +3.3V and ground wires to the breadboard.

Warning: Disconnect the VREG↔+5V wire when the LaunchPad USB cable is connected to the PC. Connect the VREG↔+5V wire when the robot is running on battery power.

5.4 System Development Plan

5.4.1 Total energy stored in a battery

In this lab you will determine total energy stored in the battery (Ni-MH).

First, charge the battery following the directions on the charger. Using the circuit shown in Figure 2, measure the current and voltage. Measure the time it takes for the voltage to drop below 80% of its nominal value. It should take hours to discharge the battery. Calculate the **energy storage** in Joules:

E = V*I*time

Also, calculate the storage in mA-hr, and compare the measured value with the specification from the manufacturer.

Recharge the batteries and measure the **noise** on the battery (without connecting it to any circuits) using the oscilloscope in AC mode. We can quantify noise either as **root mean squared voltage** (Vrms) or as **peak-to-peak voltage** (Vp-p). You will notice batteries are very low noise. However, there will be very large noises on the power once we connect it up to the robot caused by the switching regulator, the motors, and the motor drive circuits.

Lab 5: Battery and Voltage Regulation

5.4.2 Voltage Regulation

Batteries provide voltage and current. However, when operating within specifications voltage of the six NiMH batteries will range from 7.2 to 8.4V. The purpose of the **regulator** is to provide a constant voltage. The regulator on the MDPDB provides +5V output for up to 3A.

Connect the batteries to the regulator circuit on the motor board. In this lab, do not connect the LaunchPad, motors or other external circuits, just the batteries and the **MDPDB**. (You will connect the motors in Lab 12). Turn on the power and measure the battery voltage and the regulator output voltage using the voltmeter in DC voltage mode. The goal is to create a +5V regulated power source for the LaunchPad. Six fully charged NiMH batteries may have a voltage above 8V. If you have +7.2 V battery voltage, but not +5V regulated output, review the documentation for the **MDPDB**.

Once you have verified the regulator is operating properly, turn off the power, and connect the +5V regulated voltage to the LaunchPad +5V line via a current meter, as shown in Figure 6. The LaunchPad is not to be connected to the PC during these measurements. Use a voltmeter in DC mode to verify the voltage levels on the +5V and +3.3V lines. The MSP432 running at 48 MHz should draw less than 100 mA.

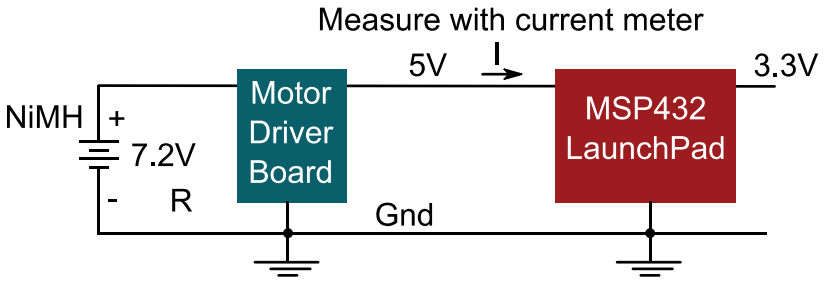


Figure 6. Block diagram of power system.

Since the +5V input to the LaunchPad is used to create the +3.3V line, the current measured on the +5V signal includes current dissipated at +5V and at +3.3V.

Use the oscilloscope in AC mode to measure the noise on the +5V and +3.3V lines.

5.5 Troubleshooting

Batteries show no voltage:

- Double check the connections.
- Check the instructions for the charger.
- Recharge the batteries.

1.2V NiMH batteries are above 1.2V:

- A fully charged NiMH may be as much as 1.4V.

LaunchPad draws more than 100 mA:

- Double check the connections.
- Above 100 mA may mean the LaunchPad is damaged

Regulator doesn't work:

- Double check the connections
- Check the datasheets for the motor board

5.6 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. The goal of this module is for you to experience voltage, current, and power as seen in resistors, capacitors, and LEDs.

- What does the regulator do?
- Does your board use a linear regulator or a switching regulator? I.e., what is the purpose of the inductor in the circuit?
- What does it mean that the regulator has 90% efficiency?
- Why does the regulator have so much noise?

Lab 5: Battery and Voltage Regulation

5.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. For example,

- If you do not have the Pololu motor board, you could build this robot, and do this lab by building your own linear regulator circuit. In particular, you could build the linear regular described in lecture using the LM7805 and two 10 µF capacitors. Since the dropout voltage of the LM7805 is 2V, you will need a battery voltage greater than 7V. The disadvantage of this approach is efficiency. When you perform the noise analysis, you will find it low noise because the LM7805 is a linear regulator. However, a 7.2 to 5V drop at 100 mA will dissipate 0.22 W in the LM7805, causing the LM7805 to get hot. 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).
- Linear regulators are inexpensive, and low noise, but they are not very efficient.
- A second option if you do not have the Pololu motor board is to build your own switching regulator circuit. In particular, you could build the switching regular described in lecture using the LM2596-5.0, a diode, an inductor, and two capacitors. Since the dropout voltage of the LM2596-5.0 is 2V, you will need a battery voltage greater than 7V. Because it is efficient, it will not get hot. Furthermore, this regulator is very student-friendly, handling overload current and even shorts to ground. Switching regulators are efficient, but they are harder to build and generate noise on the power line.
- Another way to power the robot is to use a portable cell-phone charger. A portable cell-phone charger includes a Lithium Ion battery, a charger, and +5V regulator. These systems come in various sizes and energy storage capacities. They have two USB connectors, one for charging and one for +5V power (used to charge cell phones). To use this power source, you simply plug the USB charger into the LaunchPad using a micro-USB cable. The disadvantage of this approach is the motors will need to be powered with the +5V supply. Recall power is V^2/R. If the resistance of the motor is fixed, a +5V motor voltage is only 50% of the power as compared to a +7.2V voltage (notice that 5*5 is 25, but 7.2*7.2=51.84.)

5.8 Which modules are next?

In the next few labs, we begin the process of connecting input sensors and output actuators to the microcontroller. In Module 12 we will complete the functionality of the Motor Driver and Power Distribution Board when we use the motor drive circuits to allow the software to control the two motors.

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 system complexity.
Module 12) We will interface the motors to the robot.

5.9 Things you should have learned

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

- Understand voltage, current, and energy of a battery.
- Be able to measure DC voltage, AC voltage, and current.
- Understand basic operation and purpose of a voltage regulator.
- Know how to use a voltmeter and oscilloscope,

Module 6

Introduction: General Purpose Input Output

Introduction: General Purpose Input Output

Educational Objectives:

REVIEW C programming
UNDERSTAND direction registers, input, and output
EXPLORE conversion from light to voltage, and voltage to binary
LEARN how to write software to initialize GPIO pins
DESIGN, BUILD & TEST A SYSTEM
Detect position relative to a black line on a white field

Prerequisites (Modules 1, 2, and 4)
- Running code on the LaunchPad using CCS (Module 1)
- Voltage, current, resistance, capacitance (Module 2)
- Basic C programming (Module 4)

Recommended reading materials for students:
- Volume 1 Sections 4.1 and 4.2
  Embedded Systems: Introduction to the MSP432 Microcontroller OR
  ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017
- Volume 2 Sections 2.2 and 2.4
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

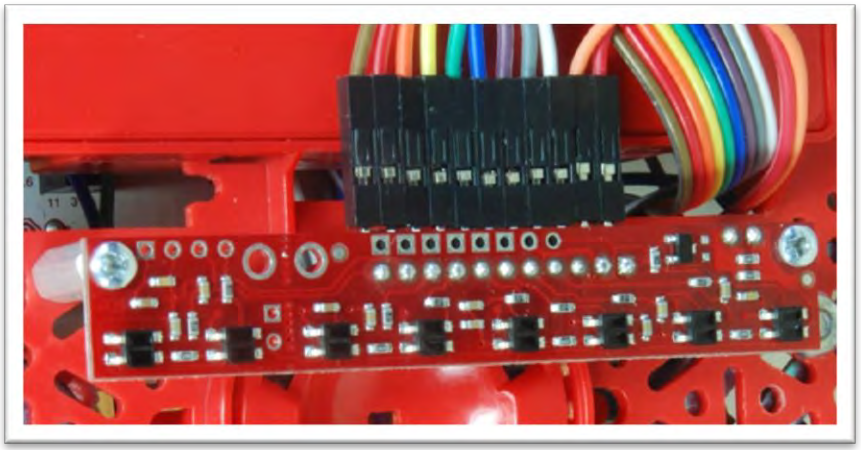


Figure 1. QTR-1RC line sensor, positioned on the bottom of the robot.

The simplest I/O port on a microcontroller is the parallel port, or **general purpose input output** (GPIO). A parallel I/O port is a mechanism that allows the software to interact with external devices. It is called parallel because multiple signals can be accessed all at once. Ports 1 – 10 are 8 bits wide meaning we read and write port pins 8 bits at time. Not every port on the MSP432 LaunchPad has all 8 pins.

An **input port** allows the software to read external digital signals. That means a read cycle access from **P1->IN** returns the values existing on the inputs of Port 1 at that time. To make a pin input, we write a 0 to its direction register. A write cycle access to an input port usually produces no effect. Input pins on some microcontrollers are 5V-tolerant, meaning input voltages can vary from 0 to 5.0 V. However, pins on the MSP432 are not 5-V tolerant, meaning the input voltages must be between 0 and 3.3 V.

While an input device usually just involves the software reading the port, an **output port** can participate in both the read and write cycles very much like a regular memory. A write cycle to **P1->OUT** will affect the values on the output pins of Port 1. To make a pin output, we write a 1 to its direction register. Since it is a readable output, a read cycle access from the port address returns the current values existing on the port pins. We can either read from **P1->OUT**, returning the values previously written, or read from the pins themselves to see the pin values, **P1->IN**.

To make the microcontroller more marketable, the ports on most microcontrollers can be software-specified to be either inputs or outputs. Microcontrollers use the concept of a **direction register** to determine whether a pin is an input (direction register bit is 0) or an output (direction register bit is 1). We define an initialization ritual as a program executed once during start up that initializes hardware and software. If the ritual makes the direction bit zero, the port pin behaves like a simple input, and if it makes the direction bit one, the port pin becomes a readable output. Each digital port pin has its own direction bit. This means some pins on a port may be inputs while others are outputs.

Introduction: General Purpose Input Output

In the lab associated with this module, you will interface a line sensor to the microcontroller, see Figure 1. Proper function of the sensor will require you to fully understand the direction register and how to perform input and output. This lab does provide an opportunity to improve your C programming skills including debugging with CCS and with an oscilloscope. Since there are measurements in this lab, you will be able to discover performance metrics such as accuracy, monotonicity, specificity, standard deviation (noise), and coefficient of variation. In previous modules, you developed code on the MSP432 using CCS, but in this module you will create a major component required to build the robot explorer. Other labs will provide additional sensors for the robot controller. In **10. Debugging** you will add bumper switches, and implement this line sensor interface using interrupts.

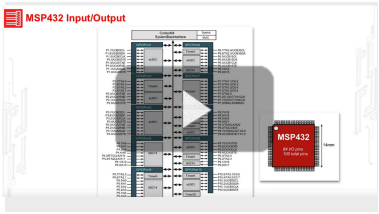
The basic approach to system development is to create components and then piece the components together to create the system. In this module, you will design develop and test the line sensor measurement required for the for robot explorer.

6. TI-RSLK Module 6 – GPIO

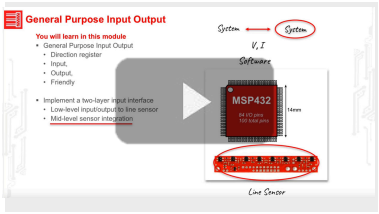
In this module, you will interface a line sensor (infra-red sensor) to the microcontroller and learn how to write software to initialize GPIO pins. The line sensor is a simple and accurate sensor for solving robotic challenges.

Optionally, download all the curriculum documents for Module 6.

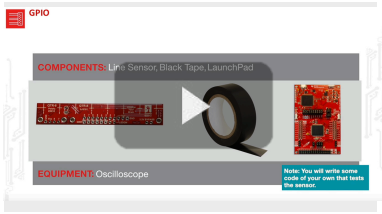
6.1 TI-RSLK Module 6 - Lecture video part I - GPIO - MSP432™
In this module you will design, develop and test the line sensor measurement required for the maze robot.



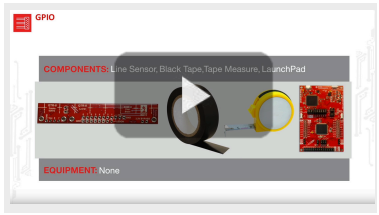
6.2 TI-RSLK Module 6 - Lecture video part II - GPIO - Programming
In this module you will design, develop and test the line sensor measurement required for the maze robot.



6.3 TI-RSLK Module 6 - Lab video 6.1 - Demonstration of how the reflectance sensor works
The purpose of this lab is to interface the reflectance sensor to the robot.



6.4 TI-RSLK Module 6 - Lab video 6.2 – Demonstration of the lab solution and testing the line sensor
In this particular portion of the lab, we're going to test the high-level performance of the line sensor.



Module 6

Lab 6 : General Purpose Input Output

Lab: General Purpose Input Output

6.0 Objectives

The purpose of this lab is to interface a line sensor that the robot will use to explore its world, see Figure 1.

1. You will learn functions, conditionals, loops, and calculations in C.
2. You will use GPIO to perform input and output.
3. You will understand how light is converted to voltage, and how voltage is converted to binary.
4. You will interface a line sensor to the microcontroller.

Good to Know: General purpose input output (GPIO) is the simplest and most pervasive means of performing I/O on the microcontroller. The sensor you interface in this lab will allow a robot to explore its world.

6.1 Getting Started

6.1.1 Software Starter Projects

Look at these three projects:
GPIO (a very simple system that outputs to four pins),
InputOutput (simple system that inputs from switches and outputs to LEDs on the LaunchPad),
Lab06_GPIO (starter project for this lab)

6.1.2 Student Resources (Links)

Meet the MSP432 LaunchPad (SLAU596)
MSP432 LaunchPad User's Guide (SLAU597)
QTR-8x.pdf, line sensor datasheet

6.1.3 Reading Materials

Volume 1 Sections 4.1 and 4.2
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 2.2 and 2.4
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller",

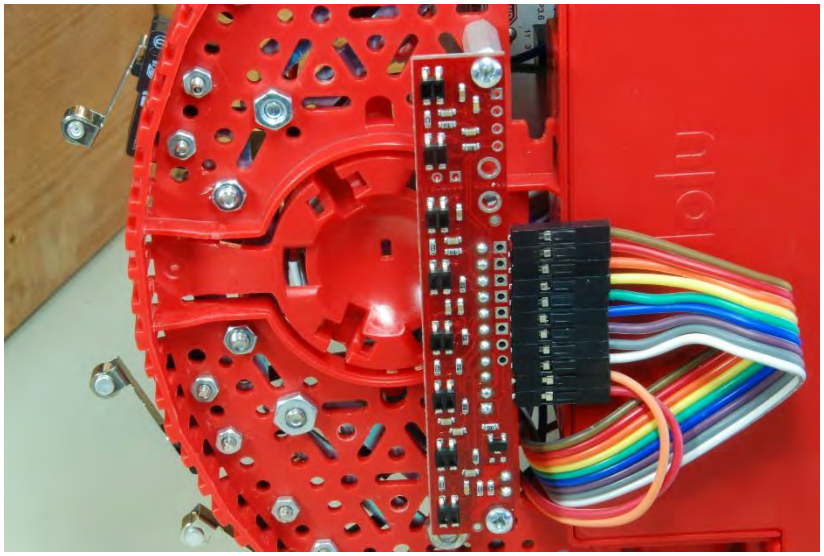


Figure 1.QTR-8RC mounted on bottom of robot, 3mm above the floor

6.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	QTR-8RC Reflectance Sensor Array	Pololu	# 961

Table 1

6.1.5 Lab equipment needed

Oscilloscope (one or two channels at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling
*Note you also need a black non reflective tape and a white surface.

Lab: General Purpose Input Output

6.2 System Design Requirements

The ultimate goal of this lab is to design a sensor system that measures the position of the robot relative to a line. For example, black tape on a white surface could be used in a robot challenge of exploring its world.

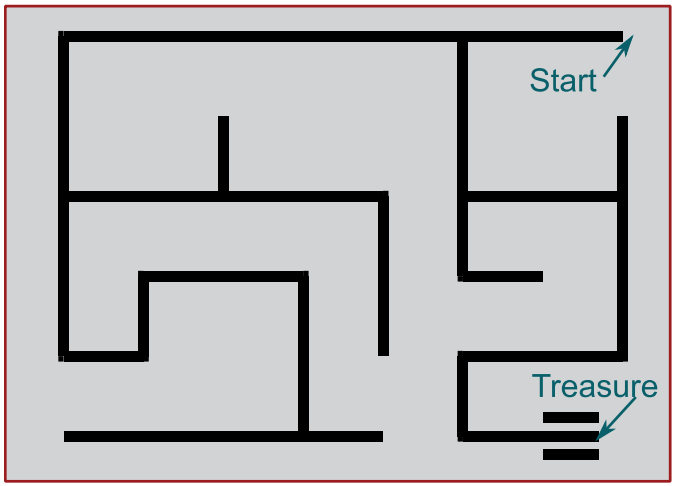


Figure 2. Possible final robot challenge of exploring the world and finding the treasure.

The robot will have eight sensors that detect the line, see Figure 3.

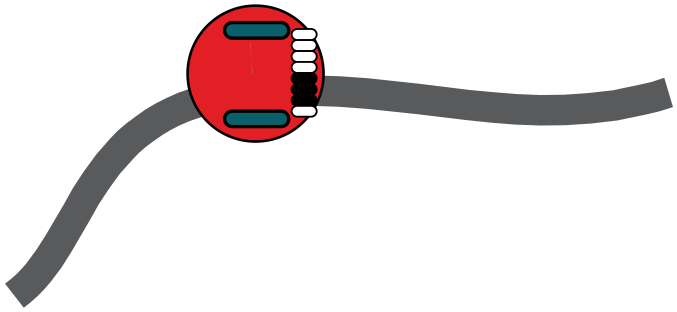


Figure 3. Robot with eight line sensors and two wheel motors.

Each sensor is binary, returning 1 if it sees black and 0 if it sees white. If the robot is properly positioned on the line, the middle sensors will see the black line. If the robot is a little off to the left or right, one or more outer sensors will see the black line. If the robot is completely off the line, all sensors will see white. We will use sensor integration to combine the eight binary reading into a single position parameter. We define **position** of the robot as the distance from the sensor to the line. The sensor we will use is about 66 mm wide (with about 9.5mm between sensors), so we should be able to estimate the robot position of -33 to +33 mm from the center of the line.

Figure 4 shows the desired output of the sensor measurement system. Basically, you are asked to create an output that varies from -330 to +330 (units 0.1mm) as a function of the position of the robot relative to the line.

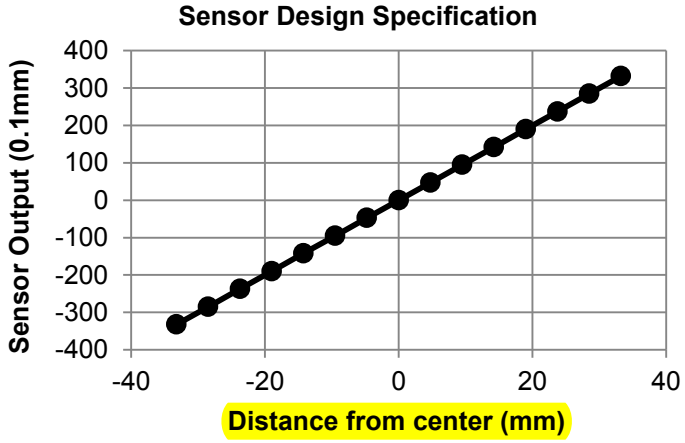


Figure 4. Desired output of the measurement system.

There are a number of performance measures with which the system could be evaluated. **Accuracy** is the difference between true line and measurement at a given time.

$$\text{Average Accuracy} = \frac{1}{N}\sum_{i=1}^{N}|x_i - T_i|$$

where x_i is the measurement and T_i is the actual or true line. Accuracy is interesting, but not very important for solving the explorer missions.

The system is **monotonic** if an increase in input never causes a decrease in output, i.e., as the input increases, the output increases or stays the same.

Similarly, as the input decreases, the output decreases or stays the same. Monotonicity will be important for the robot control system. You will determine if your system is monotonic by slowly sliding the sensor across the line and measuring the system output at each position.

Noise is also important for a control system. **Standard deviation** is quantitative measure of noise and given by:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$\sigma = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

where in this case, x_i are measurements taken with the input fixed. The **coefficient of variation** is the standard deviation divided by the mean or the average value.

$$CV = \sigma/\mu$$

Specificity is a measure of relative sensitivity of the system to the desired signal compared to the sensitivity of the measurement to other unwanted influences. A system with a good specificity will respond only to the signal of interest and be independent of these other factors. The unwanted influence you will study is angle to the line, as defined in Figure 5. In particular, you will create response curves like Figure 4, for angles 90, 75, 60, and 45 degrees.

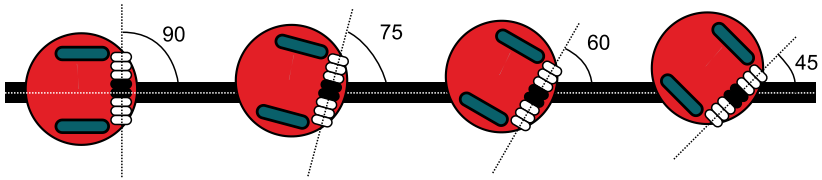


Figure 5. It is desired for the system to be able to measure relative distance to the center of the line for a wide range of angles.

6.3 Experiment set-up

The optimal sensing distance for this sensor strip is 3 mm (0.125"). **You will need to fix the distance between the line sensor and the floor to about 3 mm.** During this lab, you could use two 3 mm objects attached to each end of the sensor to set the distance and make it parallel. A better approach would be to mount the sensor on the bottom of your robot (if you have purchased the kit) again positioning the sensor parallel to the floor 3 mm above the floor. See Figure 1.

You will need access to a mockup (small representative piece) of the challenge arena (similar to shown in Figure 2) in which to test the sensor. The sensor works well on a white reflective surface, marked with black non-reflective tape.

You will implement this lab by connecting the QTR-8RC line sensor to the MSP432 LaunchPad as shown in Figure 6. Refer to the data sheet of the sensor for hook up instructions.

Warning: *TI MSP432 pins are not 5V tolerant; you must power the sensor with +3.3V.*

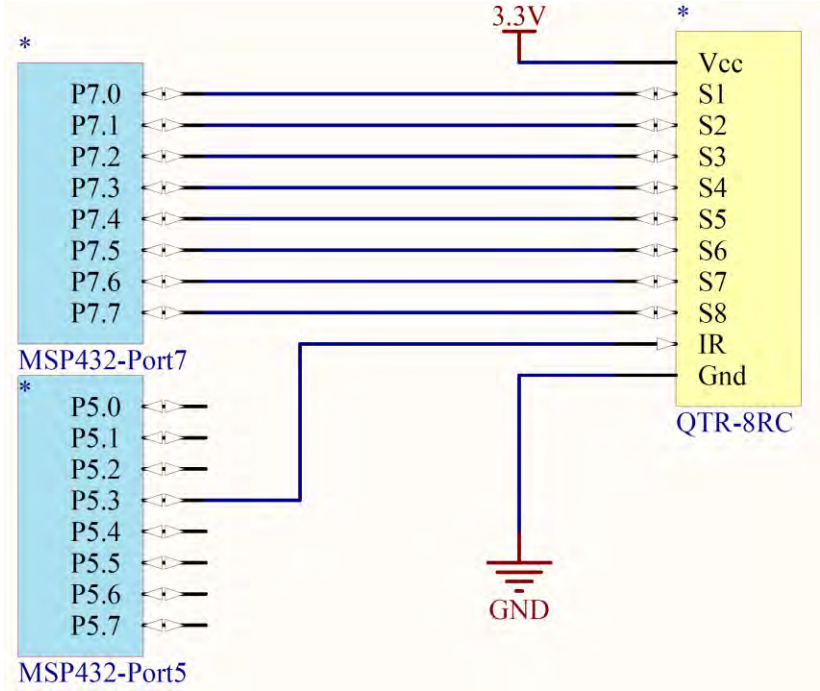


Figure 6. The 8-channel line sensor is interfaced to Ports 5 and 7. S0 (P7.0) is on the right, and S8 (P7.7) is on the left.

Lab: General Purpose Input Output

6.4 System Development Plan

6.4.1 Configuring the MSP432 ports as input and output

To initialize an I/O port for general use you will perform the following steps. First specify GPIO by writing zeros to the PxSEL0 and PxSEL1 registers. Secondly you will set the direction of the registers.

Open the **GPIO** project - Compile, download and debug this example. Using the debugger, observe the direction register and output register for Port 4 while you single step through the main program. Run the program and observe the output on Port 4 bits 3 – 0. Notice the use of a function to initialize Port 4. Is the output to Port 4 friendly or unfriendly? Notice the main2 version provides for abstraction.

Open the **InputOutput** project and notice that initialization can be done in different ways *(main, main2, and main3).* The main example is unfriendly. The *main2* example is friendly. The *main3* example is friendly and provides abstraction.

Using the **InputOutput** project, compile, download, and run the program. Using the debugger, observe the direction, input, and output registers for Ports 1 and 2 while you single step through the main program. Notice how the ports are initialized, how data are input, and how data are output. Within this project there are three versions. See comments in the project file.

Note: *The first version is "unfriendly", because it writes to the entire registers, even though it needs to affect just some of the bits. The second version is friendly, because it only alters the bits needed. The third version provides abstraction, referring to objects by their logical name (SW1IN, SW2IN, BLUEOUT, GREENOUT, REDOUT) rather than their physical implementation (P1->IN, P2->OUT).*

After following the above steps in the lab you will comment on the three versions with respect to the following:

- Ease of understanding
- Ability to integrate into larger system
- Portability, ease of implementing system on another microcontroller

6.4.2 Configuring the Low-level sensor input/output

You will begin by analyzing one of the eight sensors on the line sensor and discover how the sensor works.

You will connect the TI MSP432 ports to the line sensor QTR-8RC.

We will use Port 5, bit 3 as an output, connected to IR pin of the line sensor. P5.3 turns on the infrared (IR) LED, P7.0 is connected to one of the sensor pins of the QTR-8RC, and P1.0 is set as a digital output (used in this section for testing). See Figure 7. In particular, the goal of this section is to observe how the reflectance of the given surface affects the voltage on P7.0, and how the microcontroller converts the voltage on P7.0 into binary data depending on the color of the reflective surface.

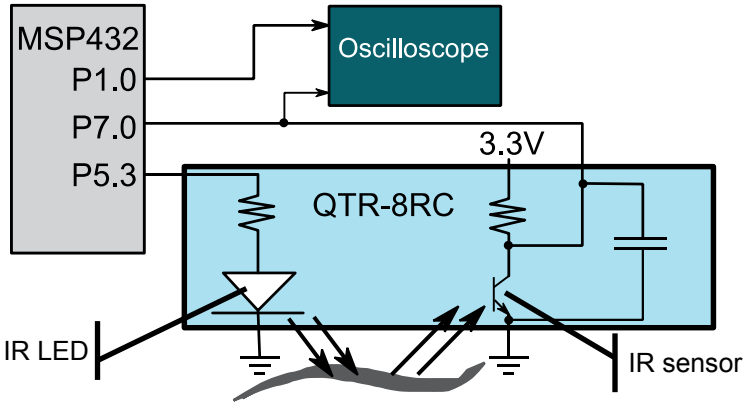


Figure 7.Hardware configuration for one sensor.

Your solutions will be placed in **Reflectance.c** so that the software will be usable for subsequence labs. See **Reflectance.h** for detailed specifications of software you will need to write. Place the sensor S1 (P7.0) 3 mm above a **white reflecting surface**. Perform this software initialization:

1) Initialize the Clock system to run at 48 MHz, **Clock_Init48MHz();**
2) Make P5.3 an output, and set it initially low
3) Make P7.0 an input
4) Make P1.0 an output (you can use any unused pin)

Lab: General Purpose Input Output

Connect P7.0 and P1.0 to a dual channel oscilloscope. If you have a single channel scope, then first measure P7.0 and then measure P1.0. Since the input (color of the line) does not change, the two separate measurements can be aligned. The body of the main program executes steps 1 to 7 over and over:

main program
  0) Initialize
while(1){
  1) Set P5.3 high (turn on IR LED)
  2) Make P7.0 an output, and set it high (charging the capacitor)
  3) Wait 10 us, **Clock_Delay1us(10);**
  4) Make P7.0 an input
  5) Run this loop 10,000 times
     a) Read P7.0 (converts voltage on P7.0 into binary)
     b) Output binary to P1.0 (allows you to see binary in real time)
  6) Set P5.3 low (turn off IR LED, saving power)
  7) Wait 10 ms, **Clock_Delay1ms(10);**
}

Notice that you read the sensor 10,000 times, looking for the input signal to change from 1 to 0. You will observe the change on P1.0 using an oscilloscope or logic analyzer

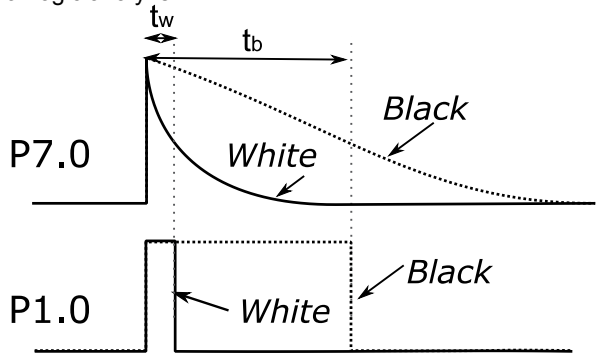


Figure 8. Conversion of light to voltage, and voltage to binary

Run the system and observe the voltage on P7.0 and P1.0 on the oscilloscope. At what voltage does the binary switch from 1 to 0 on P1.0 for the white surface? Repeat the experiment with a black non-reflective surface.

Note: *Basically, what is happening is the 10us output pulse on P7.0 charges a capacitor on the QTR-8RC board (see Figure 7). Once the microcontroller sets the pin to input, the reflected light on the light-sensitive transistor discharges the capacitor. The white reflective surface has more light on the base of the transistor, and conducts more current through the collector-emitter. This current discharges the capacitor. Notice the white surface discharges faster and P7.0 falls more quickly. Using the rise of P7.0 as a time reference of 0, measure the time P1.0 falls for the white and black surfaces, see Figure 8. For a white surface, measure the time t_w. For a black surface, measure the time t_b.*

6.4.3 Low-level sensor interface

Now that you understand how the sensor works, you will create a low-level software driver that measures all eight sensors at the same time. You will need to connect all 8 sensor inputs as shown in Figure 6. You will assume the MSP432 is running at 48 MHz clock. The initialization includes:
  1) Make P5.3 an output, and set it initially low
  2) Make P7.7 – P7.0 inputs

Create a C function that measures all eight sensors. Let **time** be a parameter passed into this function, choosing **time** between t_b and t_w. Basically, if we wait 1000 µs, then the white will have decayed to a zero, while the black will still be high. This allows us to differentiate between white and black,

Perform these 8 steps in this sequence:
  1) Set P5.3 high (turn on IR LED)
  2) Make P7.7 – P7.0 outputs, and set them high (charging 8 capacitors)
  3) Wait 10 us, **Clock_Delay1us(10);**
  4) Make P7.7 – P7.0 input
  5) Wait time us, **Clock_Delay1us(time);**
  6) Read P7.7 – P7.0 inputs (converts voltages into binary)
  7) Set P5.3 low (turn off IR LED, saving power)
  8) Return 8-bit binary measured in step 6

Passing into this function makes it easy to retune the sensor if the robot or arena changes. **Time** will be about 1000 µs, as measurements from the oscilloscope have shown. You will test your system using a simple main program similar to Program6_1 and observe the **Data** variable with the

debugger to make sure it matches expected behavior (a “0” means white and a “1” means black).

```
// Use this program to test the Read function
uint8_t Data; // QTR-8RC
int Program6_1(void){
  Clock_Init48MHz();
  Reflectance_Init(); // your initialization
  TExaS_Init(LOGICANALYZER_P7);
  while(1){
    Data = Reflectance_Read(1000); // your measurement
    Clock_Delay1ms(10);
  }
}
```

Note: *You will notice this measurement consumes all the processor time. Time is wasted while it waits the **Time**=1ms for the capacitors to discharge (or not discharge). Time is wasted again in the 10 ms delay within the main loop. Once we learn interrupts in Module 10, we can recover this wasted time.*

6.4.4 High-level sensor integration

In this section, you will combine the eight binary measurements into a single parameter representing the amount of which the robot is away from the center of the line. We will assume the sensor S1 (P7.0) is positioned on the robot's right, 33.2 mm from midline. Furthermore, we assume the sensor S8 (P7.7) is positioned on the robot's left, -33.2 mm from midline. As mentioned earlier, another goal is to make this parameter **insensitive to angle**. If the sensor is operating properly, the 8-bit binary pattern stored in **Data** falls into four categories:

1) <all 0’s> (off the line or on white surface)
2) <some 0’s, some 1’s>, e.g., 00000111 (off to the left)
3) <some 0’s, some 1’s, some 0’s>, e.g., 00110000 (over the line)
4) < some 1’s, some 0’s>, e.g., 11110000 (off to the right)

Figure 9 (source: Pololu.com) Shows the sensors are about 9.5 mm apart, with the center of the robot aligned between sensors 4 and 5 (P7.3 and P7.4)

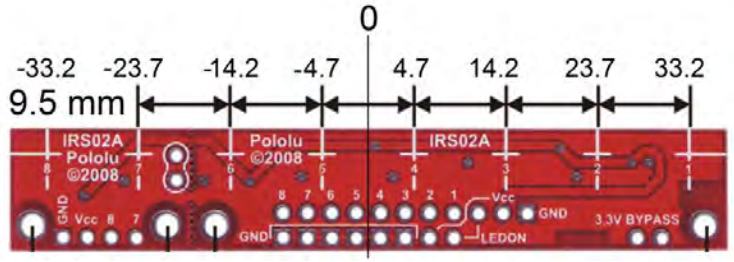


Figure 9. Position is defined as relative distance to the center of the robot.

Define an array of eight distance values in 0.1mm units, W_i for i = 0 to 7.
W = **{332, 237, 142, 47, -47, -142, -237, -332}**

Define b_i to be 0 (white) or 1 (black) for each binary bit returned by the **Reflectance_Read** function.

One possible sensor integration function is to calculate a weighted average of the eight sensor readings or binary results. Assuming there is at least one black reading, estimate distance **Reflectance_Position** as:

$$d = \frac{\sum_{i=0}^{7} b_i W_i}{\sum_{i=0}^{7} b_i}$$

In this section of the lab after connecting the MSP432 launch pad development board and the QTR-8RC, you will mount as shown in Figure 1 and then you will define true distance according to the specifications illustrated in Figure 9.

Slowly slide the sensor across the line (Figure 2 shows a maze line), and plot the measured true distance and the estimated/calculated by **Reflectance_Position**, as a function of true distance, similar to Figure 4 (first at 90 degrees as defined in Figure 5).

Comment if the response is monotonic. Interpret if there any regions where the measurement is noisy. Repeat the experiment for other angles (Figure 5) to estimate the specificity of the measurement.

Note: *The sensor will be effective if the measurements are monotonic and low noise. Another issue is **offset**. Where is the robot when the readings are 0? Since the controller will attempt to drive the distance measure to 0, this is the position to which the controller will seek.*

6.5 Troubleshooting

Sensor doesn't work:

- Check the wiring as shown in Figure 5, including power and ground
- Look at signals P5.3 and P7.0 as described in Section 6.4.2.
- Single step and verify the direction registers are correct
- Verify the computer is running at 48 MHz.
- Try another LaunchPad. Try another sensor

6.6 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.

- What is a function? How are functions used to simplify software?
- What are direction registers? How do we use them? Why do we have them?
- Is it possible to use the same pin as both an input and an output?
- What information is in the header file **Reflectance.h**? What is in the code file **Reflectance.c**? What benefits does this abstraction provide?
- What voltages does the MSP432 consider low? What voltages does the MSP432 consider high?
- What is the difference between a logic analyzer and a scope?
- What is monotonicity and why is it important?

6.7 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,

- You could use this approach to measure the position of a slide pot or joystick. Place a capacitor in parallel with the potentiometer, use the pin as an output to charge the cap, then use the pin as an input to see how long it takes for the capacitor to discharge.
- Develop alternative methods to integrate the 8 sensor readings into a single parameter that determines the relative position of the robot on the line.
- Add verification checks to **Reflectance_Position** to make sure the sensor readings are one of the four expected patterns. In particular, add checks to determine if the robot is positioned over the treasure (see the end of Figure 2.)

6.8 Which modules are next?

The GPIO pins are a simple but common means to input data into the microcontroller, or output data from the microcontroller. In addition to this line sensor, GPIO will be used in the robot for bump sensors, motor direction, LCD output, tachometer input, ultrasonic I/O, Bluetooth Low Energy (BLE), and wifi. The following modules will build on this module:

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
Module 10) Learn SysTick periodic interrupts, so these measurements occur in the background in a very time-efficient manner
Module 12) Connect the line sensor and motors to the robot.

6.9 Things you should have learned

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

- Use functions to provide software abstraction
- Perform conditionals, loops, and calculations in C
- Use the debugger to single step and visualize I/O registers
- Set the direction register for GPIO pins
- Perform friendly access to I/O registers
- Visualize an input pin converting voltage into binary
- Use an oscilloscope to visualize time behavior
- Perform incremental design and testing
- Conduct experiments to determine accuracy, monotonicity, specificity, and noise
- Integrate multiple sensor measurements into a single value

Module 7

Introduction: Finite State Machine

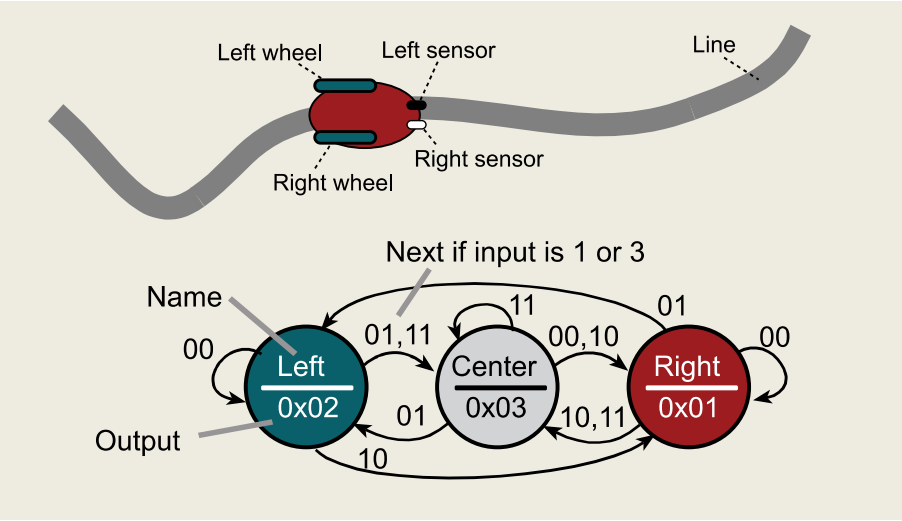
Introduction: Finite State Machine

Educational Objectives:

REVIEW C programming
UNDERSTAND variables, numbers, pointers, structures, arrays
DEVELOP debugging techniques
LEARN how to solve problems with finite state machines
DESIGN, BUILD & TEST A SYSTEM
Controller for a line tracking robot

Prerequisites (Modules 1, 4, and 6)
- Running code on the LaunchPad using CCS (Module 1)
- Basic C programming (Module 4)
- GPIO (Module 6)

Recommended reading materials for students:
- Volume 1 Sections 6.1, 6.2, 6.4 and 6.5
  Embedded Systems: Introduction to the MSP432 Microcontroller ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017
- Volume 2 Section 3.5
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017



Software abstraction allows us to define a complex problem with a set of basic abstract principles. We can then construct a system solution using these abstract building blocks. Using the abstraction gives us a better understanding of both the problem and its solution. This is because we can separate what the system does (policies) from the details of how the system works (mechanisms). This separation simplifies the design process by first describing what the system does, and then we can translate the description into a system that implements that description. Abstraction provides for a proof of correct function and simplifies both extensions and customization. The abstraction presented in this section is the Finite State Machine (FSM). The abstract principles of FSM development are the inputs, outputs, states, and state transitions. The FSM state transition graph (STG) defines the time-dependent relationship between its inputs and outputs. If we can take a complex problem and map it into a FSM model, then we can solve it with simple FSM software tools. Our FSM software implementation will be easy to understand, debug, and modify.

The problem is mapped into a well-defined model with a set of abstract yet powerful rules. Then, the software solution is a matter of implementing the rules of the model. In our case, once we prove our software correctly solves one FSM, then we can make changes to the state transition graph and be confident that our software solution correctly implements the new FSM.

Embedded systems are often deployed in safety critical systems. In these situations we must certify the solution operates exactly as intended. An abstract approach like a finite state machine (FSM) allows us to separate what it does from how it works. The complexity of a FSM is in the state transition graph, while the controller should be trivially simple. Once we certify the low-level controller is operational, we can verify the system at a high or abstract level.
In the lab associated with this module, we will use a finite state machine to create a controller for a simple line following robot. Inputs will come from two switches (simulating two line sensors) and outputs will go to two LEDs (simulating two motors on a differential drive robot). The goal of the controller is to follow the line. The purpose of this lab is to provide another lab on C programming, and serve as an introduction to robot control. In a previous module (**6. GPIO**), you interfaced an actual line sensor. Other labs will provide additional sensors for the robot controller. In **10. Debugging** you will add bumper switches. In **15. ADC** you will add IR distance sensors. In **17. Tachometer** you will add tachometers to measure wheel speed. These sensor measurements could be used as inputs to a FSM controller. In **12. DC Motors** and **13. Timers** you will interface the robot motors, which will be the outputs of the real FSM controller.

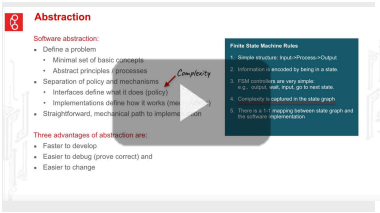
The basic approach to system development is to create components and then piece the components together to create the system. In this module, you will learn how to use FSMs as a central controller for the system.

7. TI-RSLK Module 7 – Finite state machines

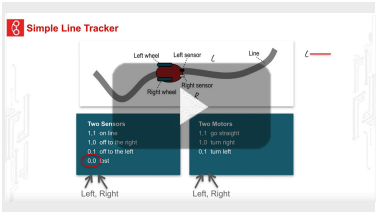
This module will demonstrate how to use finite state machines as a central controller for the system. Finite state machines are an effective design process to have in your embedded system tool box and can be used to solve problems with inputs and outputs.The basic approach to system development is to create components.

Optionally, download all the curriculum documents for Module 7.

7.1 **TI-RSLK Module 7 - Lecture video part I - Finite state machines - Theory**
In this module you will use a finite state machine to create a controller for a simple line following robot.

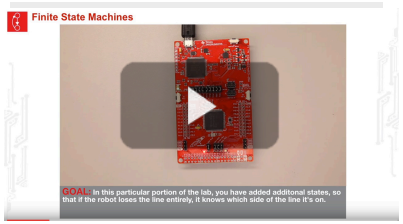


7.2 **TI-RSLK Module 7 - Lecture video part II - Finite state machine - Line tracker**
In this module, you will learn how to use finite state machines as a central controller for a robotics system.



7.3 **TI-RSLK Module 7 - Lab video 7.1 – Running the FSM starter code**
The purpose of this lab is to learn how to design a microcontroller-based finite state machine.

7.4 **TI-RSLK Module 7 - Lab video 7.2 – Running the solution code and designing a better FSM**
The purpose of this lab is to develop a line-following algorithm using a finite state machine.



Module 7

Lab 7: Finite State Machine

Lab 7: Finite State Machine

7.0 Objectives

The purpose of this lab is to develop and test a Finite State Machine (FSM) that could be used in a robot to follow a line.

1. You will learn how to use structures and pointers in C.
2. You will understand how to use FSMs to solve problems.
3. You will implement a simple line-following algorithm with an FSM.

Good to Know: Even though you will implement this lab using switches for inputs and LEDs for output, the FSM design process can be used for robot controllers. The solution to this lab will allow a robot to follow a line (black mask tape).

7.1 Getting Started

7.1.1 Software Starter Projects

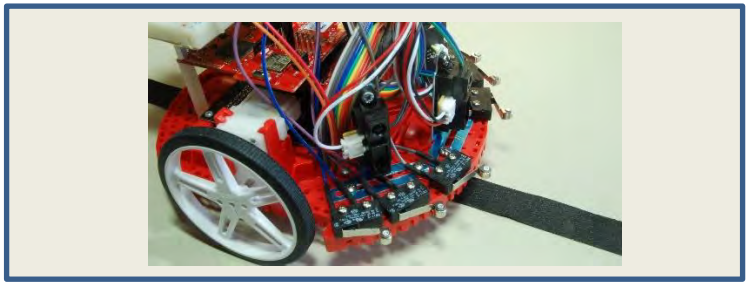
Look at these three projects:
PointerTrafficFSM (example use of a finite state machine)
LineFollowFSM (simple FSM that implements line following) and
Lab07_FSM (starter project for this lab)

7.1.2 Student Resources (in datasheets directory-Links)

Meet the MSP432 LaunchPad (SLAU596)
MSP432 LaunchPad User's Guide (SLAU597)

7.1.3 Reading Materials

Volume 1 Sections 6.1, 6.2, 6.4 and 6.5
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Section 3.5
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller",



7.1.4 Components needed for this lab

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

7.1.5 Lab equipment needed

Oscilloscope (one channel at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling

7.2 System Design Requirements

The **Lab07_FSM** starter project implements the three-state FSM shown in Figure 1, which we could use to implement a line-following robot. The 500 is the time to wait in each state in ms. On the real robot, we set these delay times to be much shorter, depending on how fast the mechanical robot responds to actuator commands. However, in this lab, the 500 ms is chosen to make it easy to see the output with our eyes.

Lab 7: Finite State Machine

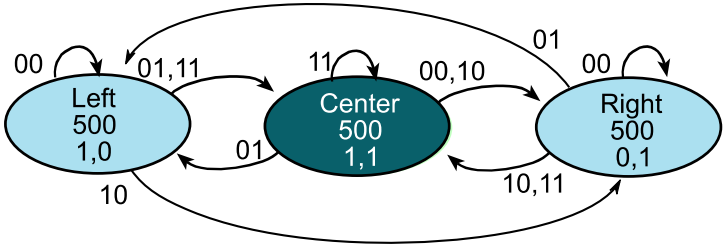


Figure 1. Moore FSM state graph to implement line following. The time in each state is shown in 1ms units.

The robot has two sensors that detect the line, see Figure 2. If the robot is properly positioned on the line, both sensors will read 1. If the robot is a little off to the left or right, one sensor reads 1, and the other sensor reads 0. If the robot is completely off the line, both sensors will read 0.

The robot has two motors, also shown in Figure 2. The two motors and a passive caster allow the robot to operate in a differential drive fashion. If the software outputs high to both motors, the robot moves forward in a straight line. If the software outputs high to just one motor, it will turn. If the software outputs low to both motors, it will stop.

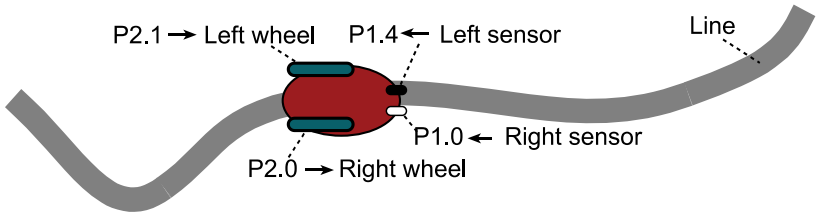


Figure 2. Robot with two line sensors and two wheel motors.

You are asked to extend this FSM, adding additional states, to implement the following behaviors.

1) The FSM in Figure 1 gets confused (has a bug) if the robot is off little bit to the left (input is 01, and the machine is oscillating between the Left and Center states) and then goes completely off the line to the left (input is 00). In this machine, if it happens to be in the Center state when it goes off the line, it will incorrectly move to the Right state even though the robot went off to the left. You will solve this problem by implementing two left states (so it oscillates between the two left states when a little left). For symmetry, you will implement two right states as well. Figure 3 shows a partial solution. If the input is 11, then the output should remain 11. If the input goes to 01 (it is a little left), then the output should toggle 1,0↔1,1 causing a slight right turn. Similarly, if the input goes to 10 (it is a little right), then the output should toggle 0,1↔1,1 causing a slight left turn.

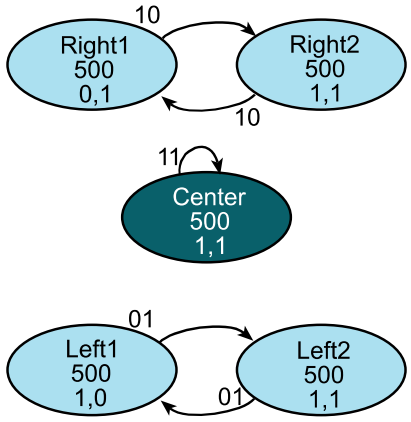


Figure 3. Expanded FSM state graph. The time in each state is shown in 1ms units.

2) The second behavior you need to implement is what happens when the robot goes completely off the line. If it goes off the line to the right (input=0,0 while in Right1 or Right2), it should make a hard left turn (output=0,1) for 5 seconds, then go straight (output=1,1) for 5 seconds. If it is still off the line at this point it should stop (output=0,0). If it finds the line, resume line following. It should take three more states to implement this behavior.

Similarly, if the robot goes off the line to the left (input=0,0 while in Left1 or Left2), it should make a hard right turn (output=1,0) for 5 seconds, then go straight (output=1,1) for 5 seconds. If it is still off the line at this point it should stop (output=0,0). If it finds the line, it should resume line following. It should take three more states to implement this behavior.

The solution should have about 11 states (5 states from Figure 3, plus 3 for lost to the right, plus 3 states for lost to the left). As long as you have 9 or more states, feel free to make assumptions or change the exact behavior of the machine. The objective of the lab is to describe the complete behavior of a system with the state transition graph, and then to implement that behavior with a very simple FSM controller. The FSM controller should have NO conditional branch statements.

Lab 7: Finite State Machine

7.3 Experiment set-up

You will implement this lab using just the MSP432 Launch Pad, without need for additional circuits, see Figure 4. The Launch Pad driver software converts the switch input to positive logic so “switch pressed” is seen as a 1, see Table 1. The LED outputs are in positive logic, see Table 2.

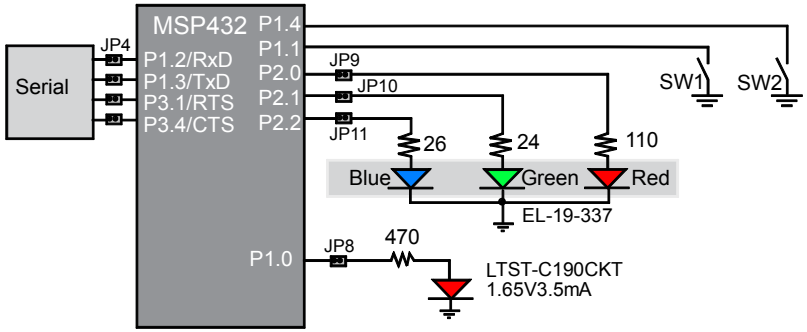


Figure 4. P1.4 is the left sensor, P1.1 is the right sensor, P2.1 is the left motor P2.0 is the right motor.

The **LaunchPad_Input** function (defined in LaunchPad.c) returns the switch position in positive logic, so pushing both switches creates an input condition of 1,1. The **LaunchPad_Output** function (defined in LaunchPad.c) sends data to the 3-bit color LED.

SW2	SW1	**LaunchPad_Input**	Meaning
pressed	pressed	1,1 = 0x03	On line
pressed	not	1,0 = 0x02	Right of line
not	pressed	0,1 = 0x01	Left of line
not	not	0,0 = 0x00	Off the line

Table 1. Switches simulate line sensors.

P2.1	P2.0	**LaunchPad_Output**	LED	Meaning
off	off	0,0 = 0x00	black	Stop
off	on	0,1 = 0x01	red	Turn left
on	off	1,0 = 0x02	green	Turn right
on	on	1,1 = 0x03	yellow	Straight

Table 2. LEDs simulate robot motor

7.4 System Development Plan

7.4.1 Line Follow FSM

The first step is to compile, download and run the **LineFollowFSM** example shown below. Using the debugger, single step through the controller (step over the functions) and observe **Input**, **Output**, and the pointer **Spt**. Notice how the structure is defined and how the pointer is used to access data in the structure. Using the debugger, determine where in memory is the FSM located (is it in RAM or ROM)?

Lab 7: Finite State Machine

```
struct State {
  uint32_t out;                // 2-bit output
  uint32_t delay;              // time to delay in 1ms
  const struct State *next[4]; // Next if input is 0-3
};
typedef const struct State State_t;

#define Center &fsm[0]
#define Left   &fsm[1]
#define Right  &fsm[2]
StateType fsm[3]={
  {0x03, 500, { Right, Left,   Right,  Center }},
  {0x02, 500, { Left,  Center, Right,  Center }},
  {0x01, 500, { Right, Left,   Center, Center }}
};
State_t *Spt;  // pointer to the current state

uint32_t Input;
uint32_t Output;

int main(void){ uint32_t heart=0;
  Clock_Init48MHz();
  LaunchPad_Init();
  TExaS_Init(LOGICANALYZER);  // optional
  Spt = Center;
  while(1){
    Output = Spt->out;            // set output from FSM
    LaunchPad_Output(Output);     // output to motors
    TExaS_Set(Input<<2|Output);   // optional
    Clock_Delay1ms(Spt->delay);   // wait
    Input = LaunchPad_Input();    // read sensors
    Spt = Spt->next[Input];       // next
    heart = heart^1;
    LaunchPad_LED(heart);         // optional
  }
}
```

In this program, this FSM performs the 4-step sequence over and over:

1) *Output* depends on *State* (LaunchPad LED)
2) *Wait* depends on State
3) *Input* (LaunchPad buttons)
4) *Next* depends on (*Input, State*)

Run the program and observe the static behavior.

i) Fill in Table 3 describing what this machine does if the input remains constant.

SW2	SW1	Input	Meaning	Output behavior
pressed	pressed	1,1	On line	
pressed	not	1,0	Right of line	
not	pressed	0,1	Left of line	

Table 3. Static response table of the simple FSM.

When just one switch is pressed, it represents the condition where the robot is a little off the line. In this situation, one wheel is active and other wheel oscillates on and off. This oscillation causes this wheel to spin, but at a slower rate. If P2.1 is high, the left wheel spins at 100%. The **duty cycle** of a digital wave is defined as the percentage of the time the signal is high. If the duty cycle on P2.0 is *n*=(high/ (high+low)), then the right motor spins at *n**100%, and the robot will gently turn. Use an oscilloscope or logic analyzer to measure the oscillation rate and duty cycle on Port P2.0. See Figure 5.

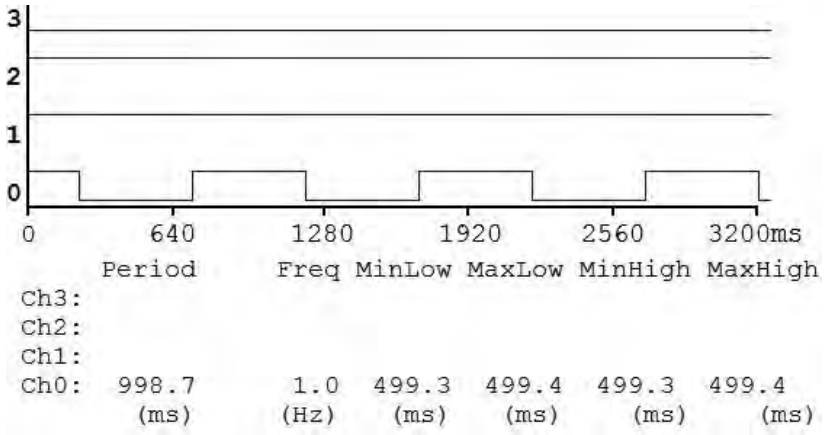


Figure 5. Logic analyzer trace showing the oscillation on the right wheel Channel 0 is 1 Hz and has a 50% duty cycle.

Note: Channel 3-2 are Input =1 (left sensor=0, right sensor =1), showing the condition a little bit off to left. Channels 1-0 are the Output (left motor=1, right motor oscillating), showing a gentle right turn.

Lastly, you will observe the bug in this FSM.
1) Start with both switches pressed (on the line);
2) Release SW2 (the robot is a little off to the left); then
3) Release SW1.

At this point you are completely off the line to the left. Repeat this 1-2-3 step sequence multiple times, and you will find sometimes it correctly ends up in the left state, but sometimes it incorrectly ends up in the right state.

7.4.2 Design an improved FSM

The second step is to design an FSM as described in the requirements section Figure 3.0. As long as your machine has 9 or more states, feel free to adjust exactly how the machine operates. In this lab section you will:

i) Draw the state transition graph

ii) Create a state transition table, and enter the C code for the data structure.
All three should be exactly the same information (no more no less). This equivalency is called **one-to-one** and it is an important feature of good FSM design. If the graph is one-to-one with the data structure in C, then we can be confident the system operates as described by the graph.

iii) You will test your system using the same 1-2-3 step sequence shown at the end of section 7.4.1. However, as long as you wait at least 500 ms with SW2 released before you release SW1, then you should always end up in one of the left states.

Perform this test at least ten times to verify it works correctly. Similarly for the right side states,
1) Start with both switches pressed (on the line);
2) Release SW1 (the robot is a little off to the right); then
3) Release SW2

At this point you are completely off the line to the right. Repeat this 1-2-3 step ten multiple times and you should always end up in one of the right states.

Use the logic analyzer to test the static behavior of the system. Assuming the input remains constant fill in Table 4.There are two off the line conditions: off to left and off to right.

SW2	SW1	Input	Meaning	Output behavior
pressed	pressed	1,1	On line	
pressed	not	1,0	Right of line	
not	pressed	0,1	Left of line	
not	not	0,0	Off the line	
not	not	0,0	Off the line	

Table 4. Static response table of the Lab FSM.

7.5 Troubleshooting

Can't program LaunchPad:

- Check the cables, jumpers on the LaunchPad development board.
- Check the Windows driver to see if the board is recognized by the operating system.
- Try another LaunchPad on this computer.
- Try this LaunchPad on another computer

Hard fault:

- Verify **Spt** always points an entry in the FSM.

Time delays are too slow or too fast:

- Verify the computer is running at 48 MHz.
- Go back and make sure Lab in Module 6GPIO still works

Lab 7: Finite State Machine

7.6 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.

- Can there be two states with the same output? Why?
- How does the FSM create the 50% duty cycle output wave? What would you change to make it 75% (even more gentle turn)? What would you change to make it 25% (sharper turn)?
- It is important that the state transition graph and the data structure in C are one-to-one. What does one-to-one mean and explain how is it true?
- This lab uses I/O abstraction in the four functions that begin with **LaunchPad_**. What information is in the header file LaunchPad.h? What is in the code file LaunchPad.c? What benefits does this abstraction provide?
- This FSM had 2 inputs. What would change if there were 3 inputs? 4 inputs?
- This FSM had 2 outputs. What would change if there were 3 outputs? 4 outputs?
- How is the FSM tested?

7.7 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,

- Replace the switch input with the actual line sensor interfaced in Lab 6. If you use the line sensor, you can expand the input from 2 bits to 4 bits.
- Use the FSM method to solve similar problems like the traffic light controller or a stepper motor controller
- This FSM used a pointer to define the current state. You could implement the FSM using an index to access the parameters of the state. E.g., `Output = fsm[index].out;`

7.8 Which modules are next?

The FSM is a powerful design tool for solving complex systems. Effective solutions to many of the possible robot challenges will include FSMs.

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
Module 10) Develop debugging techniques to prove behavior for complex systems
Module 12) Connect the line sensor and motors to the robot, and run the solution to this lab on the actual robot.

7.9 Things you should have learned

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

- Use ***struct*** to organize data
- Access data using a pointer
- Use multiple files in a project to implement abstraction
- Design a simple FSM drawing a state transition graph
- Convert a state transition graph into C data structure
- Use a logic analyzer to measure timing between inputs/outputs
- Debug the FSM and verify its proper behavior

Module 8

Introduction: Interfacing input and output

Introduction: Interfacing input and output

Educational Objectives:

LEARN Switch & LED fundamentals,
BUILD interface circuits for switches and LEDs with TI's LaunchPad development board
WRITE CODE to configure switches as inputs and LEDs as outputs
DESIGN, TEST & DEBUG A SYSTEM
"Window Intruder Detect Security System"

Prerequisites (Modules 1, 2,3, 4, 6)

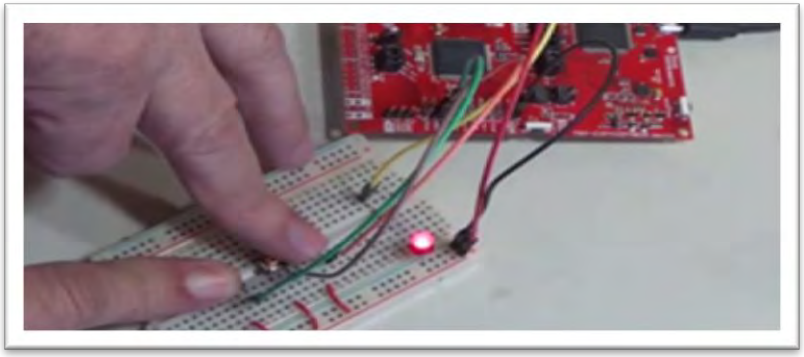
- Basic circuits, Ohm's Law (Module 2)
- Running code on the Launchpad using CCS (Module 1,4)
- GPIO (Module 6)

Recommended reading materials for students:

- Volume 1 Section 2.7, 4.1, 4.2.2, 4.3, and 4.6
  Embedded Systems: Introduction to the MSP432 Microcontroller ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or

- Volume 2 Section 2.4, and 2.6
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017



The design, development and debugging of a robotic system involves many tasks. Building of any system involves knowledge of basic components and how they work and connect with other components commonly termed as **interfacing**. In fact, we can divide all of engineering into sub-systems and interfaces. Interfaces are used to combine multiple **sub-systems** together to form a more complex system. In many examples of interfacing with a microcontroller we use input devices to feed data into a computer and output devices to allow the computer to affect its surroundings.

Another name for sub-system is **module**, which can be a software module or hardware module. Another name for a software module is **device driver**. In order to build large systems, we need a method to manage complexity. Breaking a large system into modules, which are in turn broken into smaller modules, is the standard approach to dealing with complexity. There are two aspects of a module: what it does and how it works. Modular design provides an abstraction that allows us to separate what a device does from how it works. For example, consider the red LED on your LaunchPad. There are two prototypes in the **LaunchPad.h** header file that describe what this module does:

void LaunchPad_Init(void);
void LaunchPad_LED(uint8_t data);

How the module works can be found in **the LaunchPad.c** code file. Good modularity maximizes the number of modules while minimizing the **coupling** between modules. One quantitative measure of coupling is **bandwidth**, or the amount of data flowing from one module to another.

Again, to connect modules together we need an **interface**. In the software world, one module is connected to another by the public functions that can be called. This means the header files in C define the interconnection between software modules. In the hardware world, physical devices, e.g., electrical, mechanical, chemical, biological, allow modules to be interconnected. Furthermore, when connecting software modules to hardware devices, we use a combination of software and hardware components to affect the connection. For example, the IR sensor is a hardware device that uses optics to measures distance. We will use optical devices (e.g., the sensor), electrical circuits (e.g., the analog to digital converter (ADC), and software (e.g., the ADC routines) to connect the sensor to the robotic software running on the microcontroller.

Introduction: Interfacing input and output

A common example of an input device is the **switch**. Engineers use switches in a myriad of applications touching every industry such as aerospace, automotive, chemical, communication, marine, medical, military, petrochemical, and transportation. Switches are also found in the devices we use in our homes every day. We use switches in nearly all electrical and mechanical products.
We can categorize switches as:

- Momentary pushbutton
- Rotary
- Slide switch, and
- Toggle switch.

The state of the switch is either an “OPEN” or a “CLOSE”, which can be read as binary information by a microcontroller. A typical switch has a 100-MΩ resistance when “**not pressed**” and has a 0.1-Ω resistance when “**pressed**”. We can interface switches with either positive logic or negative logic. Most commonly, the switches require the use of internal pull-up resistors for negative logic switches and internal pull-down resistors for positive logic switch interfaces. The pull-up and pull-down functions are enabled by software during initialization.

A simple example of an output device is the light emitting diode Or LED. Like switches, LEDs are binary devices, in that they can be either “ON” or “OFF”. The software controls the state of LED explicitly by calling **the LaunchPad_LED** function with a 1 or a 0. This learning module will use LEDs to report binary diagnostic information. However, we can find LED interfacing in many applications, such as optical cables, solid-state relays, digital isolation barriers, and infrared transmitters. LEDs have a nonlinear voltage current relationship. Interfacing an LED requires understanding of Ohm’s Law in resistors.

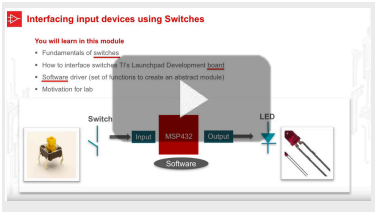
Interfacing switches and LEDs to the microcontroller is an appropriate place to start because the process is simple and the proper behavior is obvious to the learner. However, wrapped into the simple activity of connecting switches and LEDs to the microcontroller, we can expose our students to the fundamental processes of design, assembly, coding, and testing. As part of the lab, students will design **a window intruder detect system** the knowledge they gained in this module. Ultimately, the robot will use similar switches to detect an object. The students will use LEDs to provide visualizing of where and what the robot software is doing and also help with debugging, as the robotic system is put together to accomplish the planned task of solving the maze.

8. TI-RSLK Module 8 – Interfacing input and output

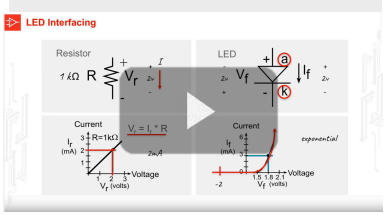
The purpose of this module is to develop interface switches and an LED so the robot can effectively detect wall collisions. Many sensors and actuators deploy LEDs, so understanding how they operate will be important to building your robot.

Optionally, download all the curriculum documents for Module 8.

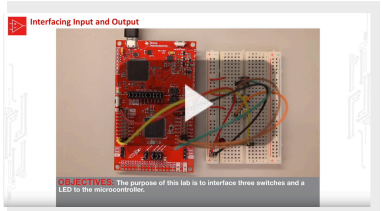
8.1 TI-RSLK Module 8 - Lecture video part I - Switches
Interfacing input and output devices using LEDs and Switches



8.2 TI-RSLK Module 8 - Lecture video part II - Interfacing input and output - LEDs
In this module you will learn the fundamentals of LEDs and switches.



8.3 TI-RSLK Module 8 - Lab video 8.1 - Interfacing switches and LEDS and debugging
The purpose of this lab is to interface three switches and a LED to the micrcontroller.



Module 8

Lab 8: Interfacing Input and Output

Lab 8: Interfacing Input and Output

8.0 Objectives

The purpose of this lab is to learn how to interface a switch and an LED to a microcontroller using TI's Launchpad development board.

1. You will first build the circuits on a breadboard and perform explicit measurements in order to verify they are operational and to improve your understanding of how they work.
2. You will then design your own **Window intruder detector alarm system** using this knowledge.

Good to Know: The switches interfaced in the lab will become bump detectors on the robot. The robot will use LED output as a debugging tool for you to visualize what the software is doing.

8.1 Getting Started

8.1.1 Software Starter Projects

Look at these example projects: **InputOutput** (input/output of switches and LEDs on the LaunchPad), **GPIO** (simple output to four pins), **Switch** (software driver of an input switch), and **Lab08_Switches_LED** (starter project for this lab)

8.1.2 Student Resources (in datasheets directory-Links)

B3F-1052.pdf Switch Datasheet
HLMP-4700.pdf LED Datasheet
CarbonFilmResistor.pdf resistor data sheet
MSP432P4xx Technical Reference Manual (SLAU356)
Meet the MSP432 LaunchPad (SLAU596)
MSP432 LaunchPad User's Guide (SLAU597)
MSP432P401R Datasheet, msp432p401m.pdf (SLAS826)

8.1.3 Reading Materials

Volume 1 Section 2.7, 4.1, 4.2.2, 4.3, and 4.6
Embedded Systems: Introduction to the MSP432 Microcontroller

Volume 2 Section 2.4, and 2.6
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller

8.1.4 Components needed for this

Quantity	Description	Manufacturer	Mfg P/N
1	**MSP-EXP432P401R LaunchPad**	**TI**	**MSP-EXP432P401R**
1	Red 2mA 5mm diffused LED	Avago	HLMP-4700
1	Carbon 1/6W, 5%, 470Ω	Yageo	CFR-12JB-470R
3	B3F tactile push button switches	Omron	B3F-1052
1	solderless breadboard	Newark	88W3961

8.1.5 Lab equipment needed

Voltmeter
Oscilloscope (one channel at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling

Lab 8: Interfacing Input and Output

8.2 System Design Requirements

In this lab you will design, develop and test a window intruder detector alarm system. As shown in the block diagram of Figure 8.1, our simple window intruder detector alarm system has three inputs and one output. The inputs are three switches are implemented with positive logic, see Figure 8.2. The first switch input is called **Activate**, which serves as the arm/disarm control. There are two window sensors, called **Window1** and **Window2**. When **Activate** is pressed or true, the security system is activated. When **Activate** is not pressed or false, the system is deactivated, meaning the alarm will be OFF regardless of the state of the window sensors. The window is in a secure position when the window sensor is pressed or true. It is unsafe if either window sensor is not pressed. The output is a LED called **Alarm,** which is implemented in positive logic. You will flash the LED at 5 Hz (on for 100ms, off for 100ms) to signify the unsafe condition. In other words, the LEDs should blink rapidly when an intruder is detected by the sensors **Window1** or **Window2**. You will connect these switches and LED to your breadboard and interface them to your MSP432 LaunchPad development board based on the truth table shown in Table 8.1.

Activate switch	Window1 sensor	Window2 sensor	Alarm (LED)
OFF	X	X	OFF
ON	Not Pressed	Not Pressed	Flash at 5 Hz
ON	Not Pressed	Pressed	Flash at 5 Hz
ON	Pressed	Not Pressed	Flash at 5 Hz
ON	Pressed	Pressed	OFF

Table 8.1. Truth Table for the Window Intruder detector alarm system.

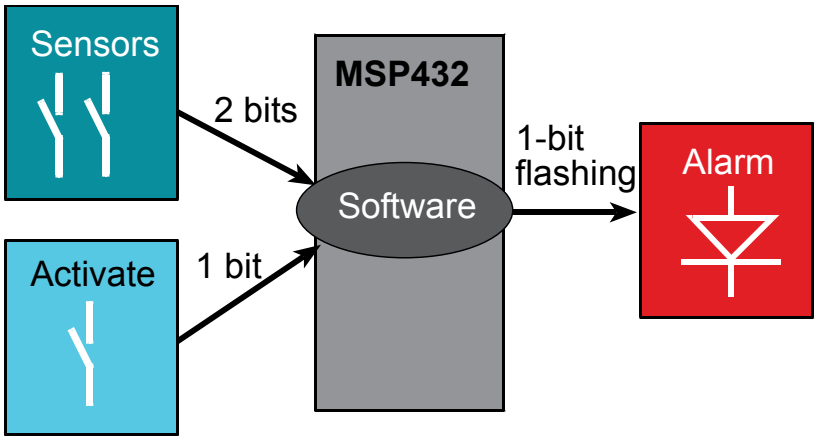


Figure 8.1. Window Intruder Detector Alarm System.

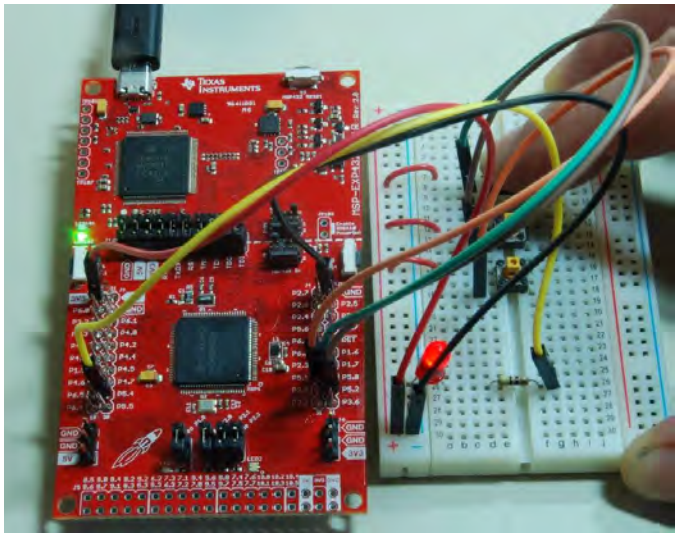


Figure 8.2. MSP432 LaunchPad and external circuits.

Lab 8: Interfacing Input and Output

8.3 Experiment set-up

8.3.1 Switch interface

You will eventually need three switches. However, you will begin by interfacing one switch to the TI LaunchPad development board. Figure 8.3 shows a possible way to connect the switch to the microcontroller. You can use an internal or external pull-down resistor to make the voltage on the pin zero when the switch is not pressed.

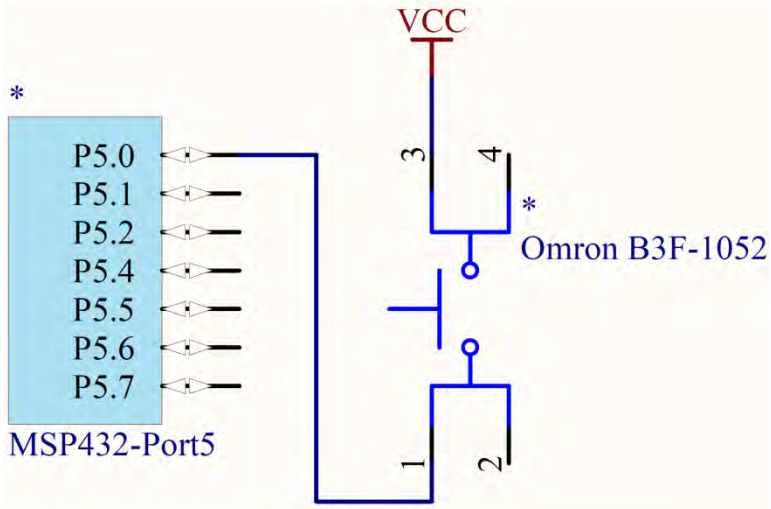


Figure 8.3 . Interface connecting the switch to P5.0 using internal pulldown. (CircuitMaker).

In order to standardize timing throughout the class, we will activate the external crystal and run at 48 MHz.

Warning: *Limit the current into and out of port pins to be less than 6 mA. One very bad way to build the switch interface is to place one side of the switch to +3.3V and the other side to ground, causing a 3.3V to ground short whenever the switch is pressed.*

Hint: Sample code to interface one switch with internal pull ups using P5.0

```
uint8_t sensor;
int Program8_1(void){
  Clock_Init48MHz();  // makes bus clock 48 MHz
  P5->SEL0 &= ~0x01;  // configure P5.0 GPIO
  P5->SEL1 &= ~0x01;
  P5->DIR &= ~0x01;   // make P5.0 in
  P5->REN |= 0x01;    // enable pull resistor on P5.0
  P5->OUT &= ~0x01;   // P5.0 pull-down
  while(1){
    sensor = P5->IN&0x01; // read switch
  }
}
```

While **Program8_1** is running, use a voltmeter to measure the voltage on the pin when the switch is not pressed and when the switch is pressed. You should get 0 V when the switch is not pressed, and you should get 3.3 V when the switch is pressed. Compare the voltage on the pin to the value in the software after the input is read. Use the debugger to observe the global variable **sensor**. You will eventually expand the system to have the three inputs, and the software will read the status of the three switches. We purposely gave this example with one switch, knowing you will need to modify the hardware and software to input from three switches. You will find a **Program8_2** in the project, which is similar to 8_1, but activates the **TExaS** logic analyzer.

Lab 8: Interfacing Input and Output

8.3.2 LED interface

Look up the desired (V_f, I_f) operating point of your LED used for this lab using its datasheet. Assuming the microcontroller pin is 3.3V calculate the resistor needed to obtain that operating point. Choose a standard resistor value near that value (e.g., 100, 220, 270, 330, 470, 680, 820, or 1000 Ω). Decide which pin you will use for the LED output, and draw the interface circuit for your LED, similar to Figure 8.4. You will have to redraw Figure 8.4 moving the LED to a pin not used by the switches.

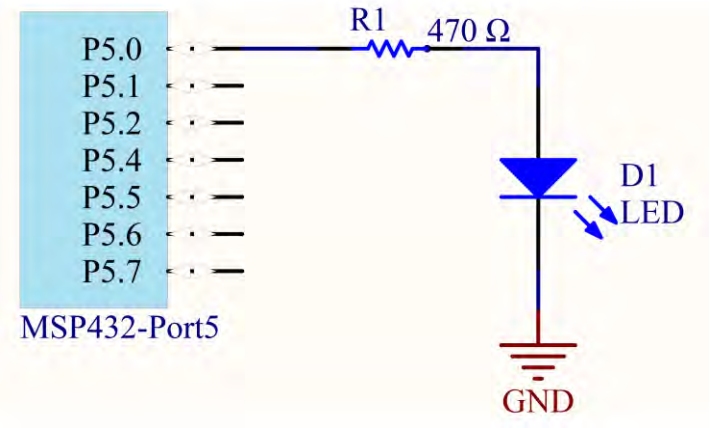


Figure 8.4. Example interface connecting the LED output to P5.4. (CircuitMaker)

Perform LED measurements with four resistance values. E.g., 220 ohms, 470, When the software outputs a high (assume 3.3 V output), estimate the current through and the voltage across the LED given the circuit you have built. However, when measuring the actual LED current and voltage, you will need to **single step**, because if you run, the pin will oscillate around a million times per second, and the LED will look dim.

Modify **Program8_3** so the microcontroller makes the appropriate pin an output. For example, if you connect the LED to P5.4, you will have to edit it so Port5 bit 4 is an output, and so the main loop oscillates bit 4. We purposely wrote **Program8_3** using the same pin as we used for input in the previous example, knowing you would need to modify this program to output to the specific port pin to which you connected your LED. The program should simply toggle the LED on and off.

Notice that the LED operations in **Program8_3** are written as a **software driver**, which is a set of functions that facilitate LED operation. Furthermore, see how the LED functions form an abstraction, separating what it does (LED init/on/off/toggle) from how it works (P5 pin 0).

Run your modified **Program8_3** and single step it until the LED is on. Measure the voltage across the resistor and the voltage across the LED. Single step the software until the LED is off and measure the two voltages again. Use Ohm's Law on the resistor to calculate the current through the resistor. The resistor current will also equal the LED current. Compare the actual (V_f, I_f) operating point of the LED with the expected values calculated during design. Measure the voltage on the microcontroller pin when the software is stopped with the output of the microcontroller is high. Compare this measured voltage to the expected value of 3.

```
Hint: Use this program to test the LED interface
void LED_Init(void){
  P5->SEL0 &= ~0x01;  // configure P5.0 GPIO
  P5->SEL1 &= ~0x01;
  P5->DIR |= 0x01;    // make P5.0 output
}
void LED_On(void){
  P5->OUT |= 0x01;  // turn on
}
void LED_Off(void){
  P5->OUT &= ~0x01; // turn off
}
void LED_Toggle(void){
  P5->OUT ^= 0x01;  // change
}
int Program8_3(void){
  Clock_Init48MHz();  // makes bus clock 48 MHz
  LED_Init();         // activate output for LED
  while(1){
    LED_On();
    LED_Off();
  }
```

Lab 8: Interfacing Input and Output

8.3.3 LED toggling

Next, you will develop and test software that flashes the LED 5 times/sec. Basically, you need to write the **Wait1ms()** function. In the next lab, we will use the SysTick timer for delays. However in this lab, you could use two *nested for loops*. Software loops are very inaccurate for time delays. So, in this lab, the delay may be any value such that the LED flashes anywhere from 4 to 6 times per second. Use a stopwatch, logic analyzer, or oscilloscope to verify the LED flashes at the desired rate.

Figure 8.5 shows Program 8.4 running with the Logic Analyzer active. To activate the logic analyzer to visualize Port 5, execute

TExaS_Init(LOGICANALYZER_P5);

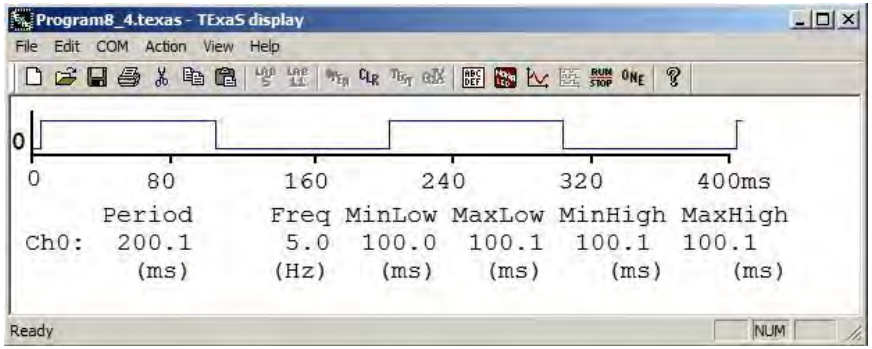


Figure 8.5. TExaS Logic Analyzer running Program 8.4 showing P5.0 toggles at 5 Hz.

```
Hint: Use this program to test the LED flashing
int Program8_4(void){
  Clock_Init48MHz();   // makes bus clock 48 MHz
  TExaS_Init(LOGICANALYZER_P5);
  LED_Init();          // activate output for LED
  while(1){
    LED_Toggle();
    Clock_Delay1ms(100);  // approximately 100 ms
  }
}
```

Lab 8: Interfacing Input and Output

8.4 Window Detector Alarm System

8.4.1 Hardware implementation:

We will use four pins on the MSP432 to implement the alarm system. It is recommended that you not use pins that already have hardware connected to them. Choose the four pins you wish to use and draw a circuit diagram of your hardware similar to Figure 8.6.

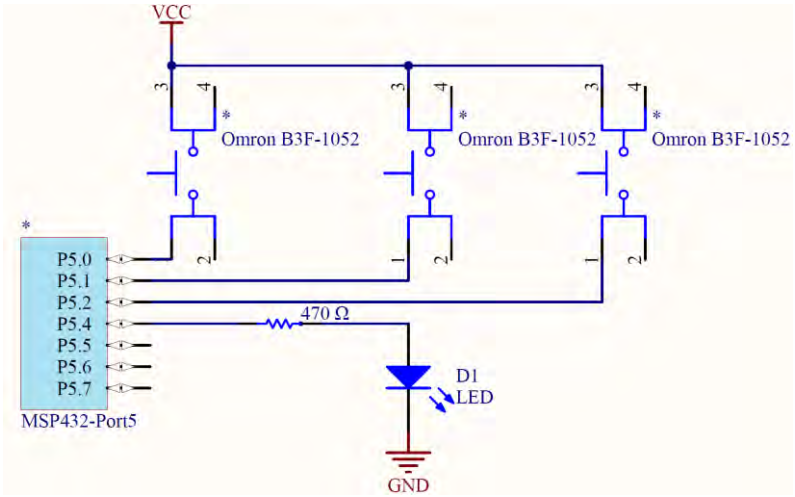


Figure 8.6 Hardware schematic interfacing input and output pins.

8.4.2 Software Implementation

The basic approach to this system is described in this pseudocode, and drawn as a flowchart in Figure 8.7.

1. Make the LED pin an output and make the three switch pins inputs.
2. The system starts with the LED off.
3. Wait about 100 ms.
4. Look at the three switches; if **Activate** is pressed and one or both **Window1** and **Window2** are not pressed, then toggle the LED once else turn the LED off.
5. Steps 3 – 5 are repeated over and over.

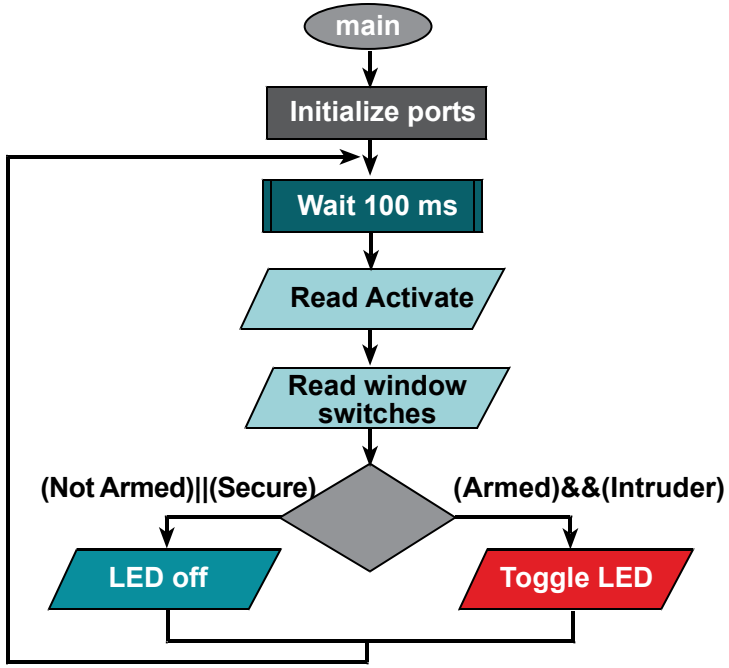


Figure 8.7. Possible software algorithm for the system drawn as a flowchart.

The structure of this lab had you build and test each subsystem separately. In this section, we combine the switch and LED interfaces to implement the Window intruder detector alarm system. To test the overall system you should first single step your software to verify it operates as intended for each of the eight cases listed in Table 8.1. Use the debugger to observe the three inputs and one output simultaneously. First, use the step over debugger command to verify the proper functional behavior

Then, you can start the program and test the system running full speed. You could use a scope or logic analyzer to verify the LED flashes at 5 Hz, when Activate is pressed and one or both Window1 and Window2 are not pressed.

8.5 Troubleshooting

Can’t program LaunchPad:

- Check the cables, jumpers on the LaunchPad development board.
- Check the Windows driver to see if the board is recognized by the operating system.
- Try another LaunchPad on this computer. Try this LaunchPad on another computer

LED does not turn on:

- Check the polarity of the LED.
- Repeat measurements done in section 8.3.2. The resistor in series with the LED should be somewhere between 220 and 2000 ohms.
- If there is 2 to 3V across the LED and the LED is dark, then it is broken (open circuit) or backwards.

Switches don’t work:

- Many switches have 4 pins, and you may be confusing across which of the pins the switch is connected.
- Use an ohmmeter measure the resistance across the switch for the pressed and not pressed conditions.
- The debugger allows you to visualize the port registers; so it is a good idea to use the debugger to check if your initialization properly configured the direction and pulldown mode .

Lab 8: Interfacing Input and Output

8.6 Things to think about

These questions are meant to test your understanding of the concepts in this lab.

- How would you make the LED brighter?
- What would happen if you plugged the LED in backward?
- What would happen if you reversed the LED and resistor? i.e., connect microcontroller output to the + side of the LED, and connect the - side of the LED to one end of the resistor, connect the other end of the resistor to ground. Would it still work? Why?
- The switch will bounce on/off/on for about 1 – 2 ms each time you push it. Similarly, the switch will bounce off/on/off when released. This lab actually debounces the switch. What operation in the main program causes the software to ignore the bouncing of the switch that occurs when touched and released? Debouncing means the software responds to the switch touch event only once, and not multiple times as the switch bounces.

8.7 Additional challenges

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

- Add a second LED to indicate if the systems is activated
- Add a green LED to indicate all is well
- Implement this lab as a finite state machine
- Add more switches and implement a digital door lock (user hits the keys in a certain order, and the lock is simulated by the LED)
- Implement a demand pacemaker. The user pushes a switch to simulate atrial sensor, and the ventricular pacing is simulated by an LED.
- Modify the following Challenge function so the system removes switch bouncing and properly counts the number of times the switch is pressed.

```
int Challenge(void){ uint32_t Count=0;
  Clock_Init48MHz();  // makes bus clock 48 MHz
  Switch_Init();      // activate input from switch
  while(1){
    while(Switch_Input()==0){}; // wait for touch
    Count++;  // number of times switch is touched
    while(Switch_Input()!=0){}; // wait for release
  }
}
```

Lab 8: Interfacing Input and Output

8.8 Which modules are next?

In this section, we list future modules that build on the concepts learned in this module.

Module 9) Use SysTick to implement time delay and dim an LED
Module 10) Add bump sensors to robot using switches
Module 13) Use periodic interrupts to run tasks in the background
Module 14) Use interrupt triggered to recognize a switch has been pressed

8.9 Things you should have learned

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

- Use an ohmmeter, voltmeter and logic analyzer
- Draw a circuit using a program like CircuitMaker
- Build circuits using a breadboard
- Program the direction register
- Perform input/output using a digital port, writing the functions as a software driver
- Interface a positive logic switch with pulldown
- Interface a positive logic low-current LED
- Create software delays using for-loops
- Toggle an LED using software delays

Module 9

Introduction: SysTick Timer

Introduction: SysTick Timer

Educational Objectives:

LEARN SysTick timer fundamentals
USE SysTick to generate accurate time delays
LEARN how to measure pulse times, and period with a logic analyzer
LEARN how to measure amplitude and period with an oscilloscope
CREATE an analog low pass filter (LPF) using an RC circuit
USE PWM and an LPF to create a digital to analog converter (DAC)
DESIGN, TEST & DEBUG A SYSTEM
Control the brightness of an LED using PWM

Prerequisites (Modules 5, 7, and 8)
- Voltage, current, resistor, capacitor (Module 5)
- Microcontroller GPIO (Module 7)
- Switch and LED interfaces (Module 8)

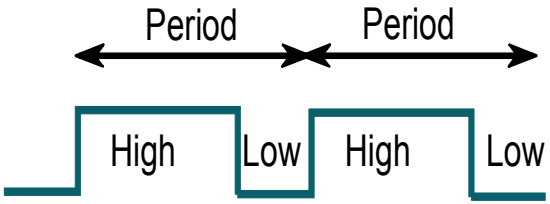
Recommended reading materials for students:
- Volume 1 Sections 4.4 and 8.7
  Embedded Systems: Introduction to the MSP432 Microcontroller ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or

- Volume 2 Sections 2.6, and 6.3
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

Time is an important parameter for an embedded system. As an **input**, measuring time includes measuring frequency, period and pulse width. For example, in the GPIO module, we saw the optical reflectance of the line sensor translated to a voltage versus time response, and the microcontroller converted this sensor data into digital form by measuring the length of time it took for the response to change from logic high to logic low. .

As an **output**, the microcontroller will create signals that affect its environment. In the **8. Switches and LED** module we needed to manage time in order to oscillate the LED at 5 Hz. In this module, we introduce the **pulse width modulation** (PWM), which is a method using time to deliver an adjustable power to a device. With PWM, the software generates a digital output of fixed frequency. Let *Period* be the fixed period of this digital wave, let *High* be the time the signal is high, and let *Low* be the time the signal is low. Typically, when the signal is high, power is applied to the external device. The software adjusts the high and low times, such that *Period=High+Low* is fixed. In many systems, the delivered power is linearly proportional to the **duty cycle**, *High/(High+Low)*.



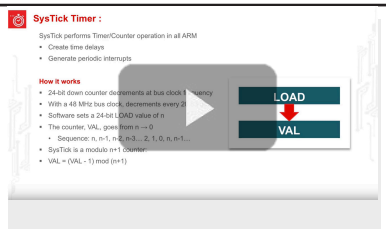
In the lab associated with this module, we will use PWM to dim the brightness of an LED. By passing the PWM output to an **analog low pass filter**, with one resistor and one capacitor, we can create a digital to analog converter (**DAC**). Using the RC filter at this point is a good way to explain how motors respond to the PWM wave. In a future module (**12. DC Motors**), our software uses the software generated PWM to control power to a motor. PWM generation is so important to embedded systems, we will show you in the **13. Timers** module how to create multiple waveforms off-loading the waveform generation into hardware. In this approach, the software still sets the duty cycle and period, but the timer hardware does the work of generating the digital waves.

9. TI-RSLK Module 9 – SysTick timer

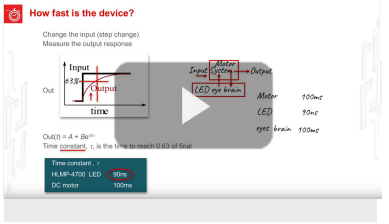
In this module, you will learn the fundamentals of SysTick timers and pulse width modulators (PWM), including how to measure pulse times and period with a logic analyzer and amplitude with an oscilloscope. It is important to understand the concept of PWM as we will use it to adjust power to the motors.

Optionally, download all the curriculum documents for Module 9.

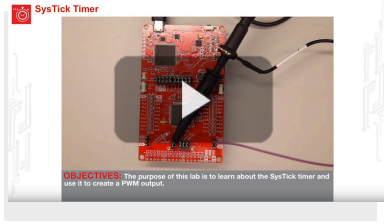
9.1 **TI-RSLK Module 9 - Lecture video part I - SysTick Timer - Theory**
In this module you will learn SysTick timer fundamentals.



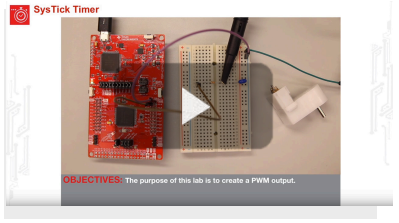
9.2 **TI-RSLK Module 9 - Lecture video part II - SysTick Timer - PWM**
You will learn the concept of Pulse Width Modulation (PWM) and duty cycle.



9.3 **TI-RSLK Module 9 - Lab video 9.1 – Demonstrating running heartbeat by adjusting the duty cycle**
The purpose of this lab is to learn about the SysTick timer and use it to create a PWM output.



9.4 **TI-RSLK Module 9 - Lab video 9.2 – Demonstrate running sine wave output to adjust power**
The purpose of this lab is to create a PWM output.



Module 9

Lab 9 : SysTick Timer

Lab: SysTick Timer

9.0 Objectives

The purpose of this lab is to learn how to use the SysTick timer to manage time.

1. You will first implement an accurate time delay.
2. You will then use the time delay to create a PWM output.
3. With a hardware low pass filter, you will use the PWM to implement a DAC.

Good to Know: Timers, like SysTick, are used in the robot to manage time. SysTick will be used to execute tasks on a periodic basis. The ultrasonic sensors used in the robot calculates distance by measuring the time it takes for sound to travel, reflect off a surface, and return back to the sensor. The concept of PWM will be used to apply a variable power to the robot motors.

9.1 Getting Started

9.1.1 Software Starter Projects

Look at these two projects:
SysTick (example use of the SysTick timer),
Lab09_SysTick (starter project for this lab)

9.1.2 Student Resources (in datasheets directory)

MSP432P4xx Technical Reference Manual (SLAU356)
Meet the MSP432 LaunchPad (SLAU596)
MSP432 LaunchPad User's Guide (SLAU597)
MSP432P401R Datasheet, msp432p401m.pdf (SLAS826)
CarbonFilmResistor.pdf, resistor datasheet
CeramicCapacitor.pdf, capacitor data sheet

9.1.3 Reading Materials

Volume 1 Sections 4.4 and 8.7
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 2.6 and 6.3
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller",

9.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	Ceramic capacitor, 0.47 µF	Kemet	C320C474M5U5TA
1	Carbon 1/6W, 5%, 470 Ω	Yageo	CFR-12JB-470R
1	solderless breadboard	Newark	88W3961

9.1.5 Lab equipment needed

Voltmeter
Oscilloscope (one channel at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling

9.2 System Design Requirements

In the first part of the lab you will generate a heartbeat wave using the red LED on the TI Launchpad Development kit. You will then use the concept and generate a PWM DAC.

The LED will oscillate from bright to dim to-off to dim almost exhibiting a sine wave, so the LED "looks" like it is breathing. This lab will use the two switches on the LaunchPad to activate and deactivate the heartbeat.

- The heartbeat activates (and continues indefinitely) when the operator pushes SW1
- The heartbeat deactivates when the operator pushes SW2

The operator may push the switches multiple times, and the heartbeat should start and stop as described above. If you can ignore the start button while the heartbeat is active, and ignore the stop button while the heartbeat is inactive.

Lab: SysTick Timer

The basic idea is to use your SysTick wait function to create a digital output signal on P1.0. When active, the period of this signal should be fixed at 10,000 µs. However, software will adjust the duty cycle as a means to control the brightness of the LED. Let *H* be the time the LED is on and *L* be the time the LED is off. The software will guarantee that *H+L* is always 10000 µs. However, when active, *H* will vary from 100 to 9900. The duty cycle is defined as

$$Duty = H/(H+L) = H/10000$$

The brightness of the LED is linearly related to the duty cycle. To give your heartbeat flair, you will oscillate the duty cycle sinusoidally, as illustrated in Figure 1. When the duty cycle is large the LED will be bright, when the duty cycle is 50% the LED will be dim, and when the duty cycle is low the LED will be off.

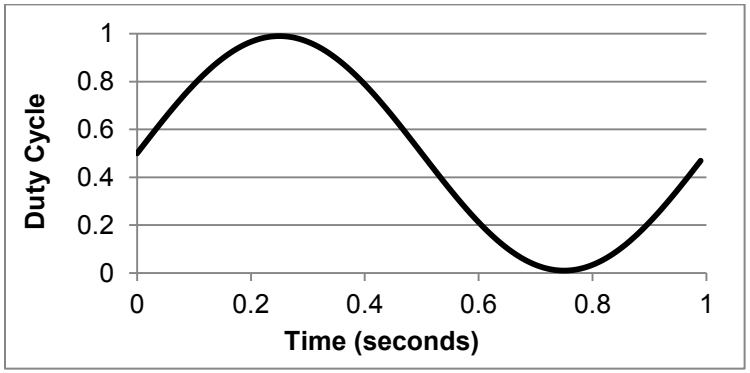


Figure 1. Plot of Duty=H/10000 as a function of time.

Figure 2 shows the MSP432 LaunchPad. You will use the red LED connected to P1.0. You will use switch 1 (SW1) connected to P1.1 and switch 2 (SW2) connected to P1.4.

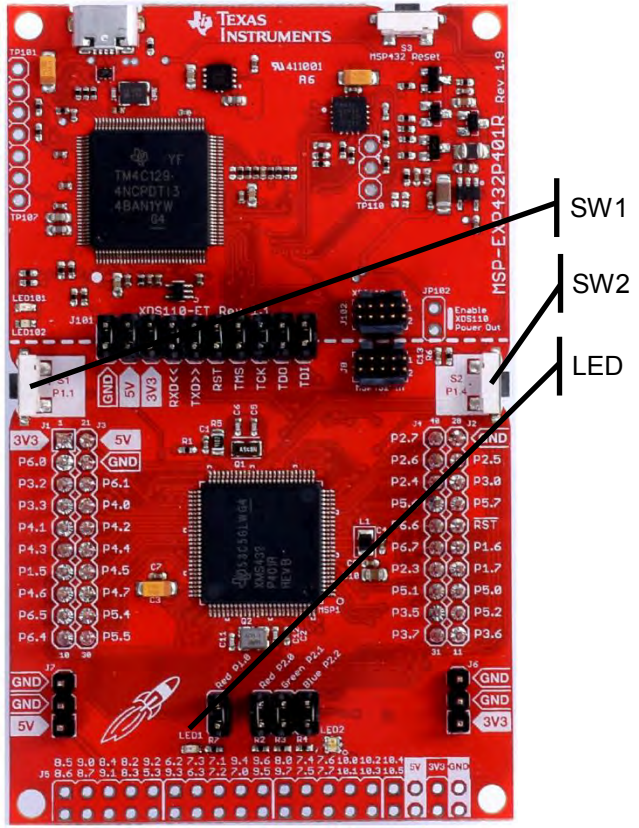


Figure 2. The LaunchPad without external circuits are used for this lab.

Lab: SysTick Timer

9.3 Experiment set-up

9.3.1 Hardware for periodic heartbeat

The LED breathing will be implemented with the MSP432 LaunchPad, without need for additional circuits, see Figure 3.

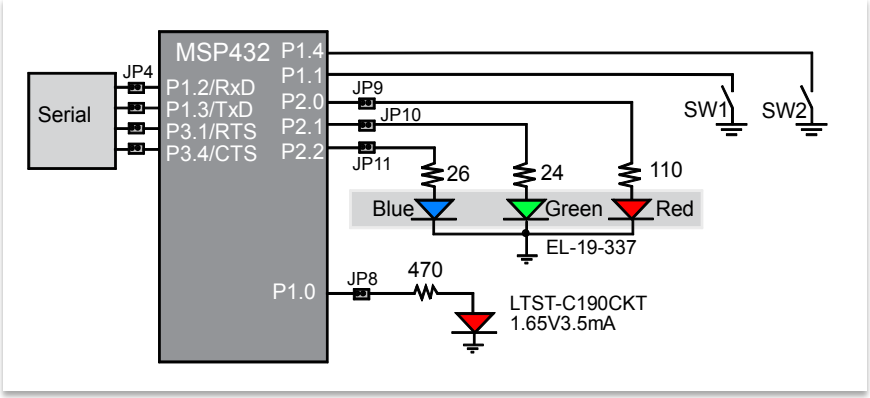


Figure 3. Create the periodic heartbeat on P1.0.

9.3.2 Hardware for PWM DAC

To implement the PWM DAC, you will need to build an analog low pass filter. The voltmeter and oscilloscope should be connected across the capacitor, see Figure 4.

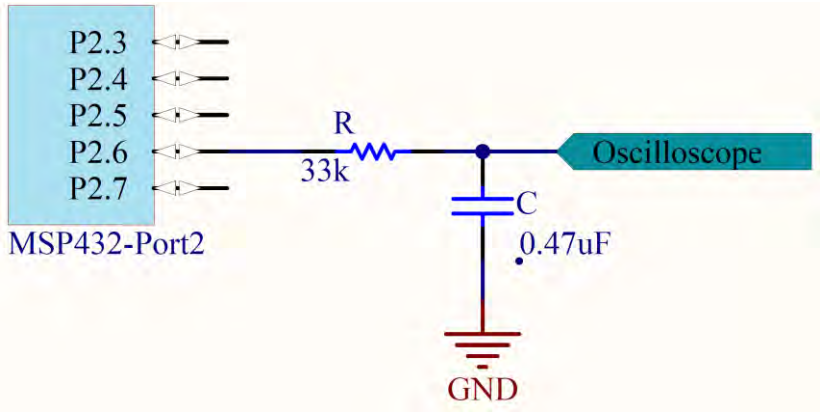


Figure 4. Use an external passive 10 Hz analog low pass filter to convert the PWM signal (P2.6 in this case) into a DAC analog output voltage.

9.4 System Development Plan

9.4.1 SysTick Wait

The first step is to write, develop and test the SysTick wait function.

The following is a software driver function that initializes **SysTick**. In this lab, we will not use interrupts. This initialization function is called once at the beginning of the main program, but before the software uses **SysTick**.

The prototype for this function is:

```
void SysTick_Wait1us(uint32_t delay);
```

where **`delay`** is the prescribed time to wait in µs. You may assume **`delay`** is greater than 2 µs and less than 349,000 µs.

```
// SysTick Initialization
void SysTick_Init(void){
  SysTick->LOAD = 0x00FFFFFF;  // maximum reload value
  SysTick->CTRL = 0x00000005;  // enable, no interrupts
}
```

Lab: SysTick Timer

The sequence of steps for the SysTick wait function are:

1. Write a desired value into the SysTick **LOAD** register
2. Clear the **VAL** counter value, which also clears the **COUNT** bit
3. Wait for the **COUNT** bit in the SysTick **CTRL** register to be set

Program9_1 shows how to test the wait function by creating a 75% duty cycle digital output. Use a logic analyzer or oscilloscope to verify the proper timing of the wait function. The signal should be high for about 7.5 ms, and low for 2.5 ms.

```
int Program9_1(void){
  Clock_Init48MHz();  // makes bus clock 48 MHz
  SysTick_Init();
  LaunchPad_Init();   // buttons and LEDs
  TExaS_Init(LOGICANALYZER_P1);
  while(1){
    P1->OUT |= 0x01;   // red LED on
    SysTick_Wait1us(7500);
    P1->OUT &= ~0x01;  // red LED off
    SysTick_Wait1us(2500);
  }
}
```

9.4.2 Generate a PWM Output

The second step is to extend the operation to implement digital waves with a sinusoidally-varying duty cycle. For example, if ***H*** = 5000, then ***L*** will be 5000, and the LED will have 50% brightness. Alternately, if ***H*** = 100, then ***L*** will be 9900, and the LED will have 1% brightness. To output a wave with fixed frequency and with fixed duty cycle, the main loop will implement these four steps in this order, over and over

1. Set P1.0 high
2. Wait ***H*** µs using your **SysTick_Wait1us** function
3. Clear P1.0 low
4. Wait ***L*** µs using your **SysTick_Wait1us** function

PulseBuf is a ROM-based table consisting of 100 pulse-times, in units of µs, which constitute a sinusoidally-varying duty cycle. Because 100*10 ms is one second, one way to create the sinusoidally-varying heartbeat is execute the following sequence over and over. If you execute steps 1 – 7 over and over again, each time through the loop using a new *H* value, the LED will flash at 1 Hz.

1. Look up a new ***H= PulseBuf [i]*** value
2. Calculate ***L*** = 10000-***H***
3. Set P1.0 high
4. Wait ***H*** µs using your **SysTick_Wait1us** function
5. Clear P1.0 low
6. Wait ***L*** µs using your **SysTick_Wait1us** function
7. ***i*** = ***i*** +1, if ***i*** ==100, roll back to ***i*** =0
8.

```
// Array used in this lab to create sine wave
const uint32_t PulseBuf[100]={
    5000, 5308, 5614, 5918, 6219, 6514, 6804, 7086,
    7361, 7626, 7880, 8123, 8354, 8572, 8776, 8964,
    9137, 9294, 9434, 9556, 9660, 9746, 9813, 9861,
    9890, 9900, 9890, 9861, 9813, 9746, 9660, 9556,
    9434, 9294, 9137, 8964, 8776, 8572, 8354, 8123,
    7880, 7626, 7361, 7086, 6804, 6514, 6219, 5918,
    5614, 5308, 5000, 4692, 4386, 4082, 3781, 3486,
    3196, 2914, 2639, 2374, 2120, 1877, 1646, 1428,
    1224, 1036,  863,  706,  566,  444,  340,  254,
     187,  139,  110,  100,  110,  139,  187,  254,
     340,  444,  566,  706,  863, 1036, 1224, 1428,
    1646, 1877, 2120, 2374, 2639, 2914, 3196, 3486,
    3781, 4082, 4386, 4692};
```

Use an oscilloscope or logic analyzer to test your solution.

Notice, however, that this method of creating a PWM output will require all of the processor's attention. Once we start putting the modules together on the robot, we will create PWM outputs using hardware timers (in the **Timers** module) so the PWM generation will not require exclusive attention of the software. For now, however, the goal is to simply understand PWM.

9.4.3 Add Switch Functionality

The third step is to add the switch functionality; such that one switch starts the heartbeat and another switch stops the heartbeat. Switch bouncing does not matter in this lab, because you can ignore the start button while the heartbeat is active, and ignore the stop button while the heartbeat is inactive.

9.4.4 Create PWM DAC

The fourth step is to design a digital to analog converter using the PWM output. There are two motivations for this section. First, a DAC is inherently useful device, and using PWM to implement a DAC provides for low-cost, high-resolution implementation for signals less than 1 kHz. Second, the RC circuit in this section mimics the behavior of the motor, so we can consider the voltage output of this circuit to be analogous to the power delivered to the motor.

Recall the frequency of the digital wave is 100 Hz. In the PWM method, the frequency will be fixed. The LED is indeed fast enough to respond on and off to this wave that we have created. Look in the data sheet for the HLMP-4700 LED. You will find it has a response time of 90 ns. So, while running at 100 Hz, the LED will completely turn on and completely turn off.

However, our eyes cannot detect waves at 100 Hz, which is why our eyes perceive the 75% duty wave at 75% brightness. We can use PWM to control other devices that respond slowly as compared to the 100 Hz wave. If the time constant of the device is slow compared to the PWM frequency, the device responds to the average signal ($H/(H+L)$) and not the instantaneous on and off. To see this powerful method of PWM in another example, we need to move the output to an unused pin, so the pin is not connected to any LED circuits. An example of a slow device is an analog low pass filter implemented with a resistor and capacitor, as shown in Figure 4. The cutoff frequency of the filter will be

$$f_c = 1/(2\pi RC)$$

To make this work, we need 1 Hz < f_c < 100 Hz, so the circuit passes the 1 Hz wave and rejects (or smooths) the 100 Hz wave. In fact, our eyes have a cutoff at about 10 Hz. So, we will choose

$$RC = 1/(2\pi 10\text{Hz}) \approx 0.016 \text{ sec.}$$

One possible combination is R=33 kΩ, and C = 0.47 μF. It also works at R = 3.3 kΩ, and C = 4.7 μF.

Warning: Choose a resistor value larger than 3.3 kΩ, in order to restrict the current below 3.3V/3.3 kΩ = 1 mA. Furthermore, we suggest choosing a resistor value much less than the input impedance of your oscilloscope probe.

The static test of your PWM-implemented digital to analog converter uses a voltmeter. Connect the voltmeter across the capacitor in Figure 3. **Program9_2** implements a 100-Hz wave with known duty cycle, ($H/(H+L)$) on P2.6.

```
int Program9_2(void){
  uint32_t H,L;
  Clock_Init48MHz();  // makes bus clock 48 MHz
  SysTick_Init();
  TExaS_Init(SCOPE);
  P2->SEL0 &= ~0x40;
  P2->SEL1 &= ~0x40; // 1) configure P2.6 as GPIO
  P2->DIR |= 0x40;   // P2.6 output
  H = 7500;
  L = 10000-H;
  while(1){
    P2->OUT |= 0x40;   // on
    SysTick_Wait1us(H);
    P2->OUT &= ~0x40;  // off
    SysTick_Wait1us(L);
  }
}
```

Run this program for five different duty cycles and plot the DC voltage as a function of duty cycle. Your data should look similar to Figure 5.

Lab: SysTick Timer

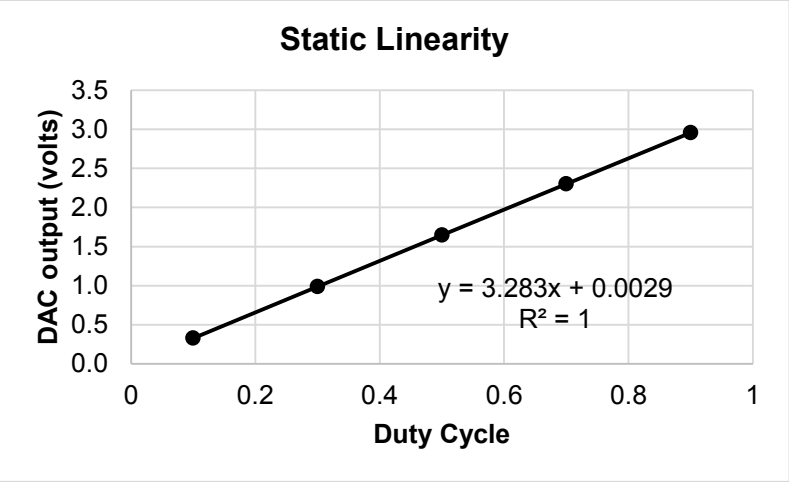


Figure 5. Example measurement data showing DAC linearity.

The above graph shows the relationship between DAC output voltage and duty cycle. Given this implementation of PWM the number of different duty cycles you can create is called the **precision** (with units of alternatives). Typically we define precision in bits.

$$Bits = \log_2(Alternatives)$$

Theoretically, if there are 10,000 alternatives, the equivalent number of bits of this DAC is approximately 13.

Resolution is the smallest change in DAC voltage that can be created. If the H were to increment by 1, In this case, if the H is incremented by 1, then the DAC analog output be (theoretically) increase by 3.3V/10000 = 0.33 mV.

Range is the maximum voltage (3.3V) minus the minimum voltage (0V). Notice that the range (in V), precision (in bits), and resolution (in V) are related.

$$Range = 2^{Precision} * Resolution$$

Next, let's measure the actual system performance of the circuit built from Figure 4 and compare actual to theoretical value.

Test (i) Using **Program9_2**, set H=9000 and L=1000. This will set the duty cycle to 90%. Then using a voltmeter measure the DC voltage of the DAC. The voltage on the capacitor should be about 0.9*3.3V. Let *S* (signal) be this DC voltage measurement.

Next, without changing the duty cycle, change the voltmeter setting and measure the AC voltage of the DAC. Let *N* (noise) be this measurement in volts. Calculate signal to noise ratio as *SNR = S/N*. In this measurement, we define the RMS AC voltage as the resolution of the DAC. Similarly, we approximated DAC range as the value at 90%.Therefore, the equivalent number of bits considering noise is

$$Precision\ (bits) = \log_2 S/N$$

Test (ii) For the same circuit as shown in Figure 4, we will use an oscilloscope. Connect the scope probe across the capacitor. Now run your sinusoidally-varying duty system and observe the output on the scope. Figure 6 shows a typical analog output, measured with the TExaS oscilloscope. Figure 6 shows the filter does not remove all the 100 Hz components; it does pass the 1 Hz, but also passes some of the 100 Hz. There is a large 100-Hz component in the signal arising from the PWM signal. If your scope has a spectrum analyzer function, you can use it to see the amplitude at 100 Hz, caused by the PWM frequency.

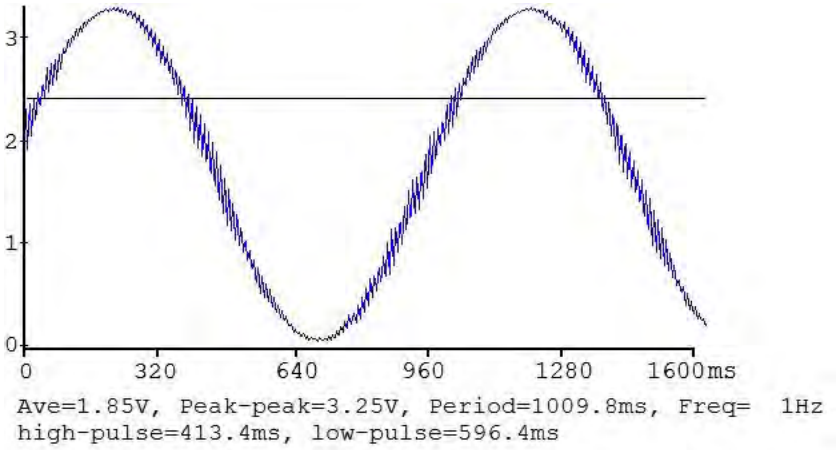


Figure 6. Example analog output of the PWM DAC.

9.5 Troubleshooting

Can't program LaunchPad:

- Check the cables, jumpers on the LaunchPad development board.
- Check the Windows driver to see if the board is recognized by the operating system.
- Try another LaunchPad on this computer. Try this LaunchPad on another computer

SysTick delay is not correct:

- Make sure the MSP432 clock is running at 48 MHz.
- Make sure all the integers used in the SysTick functions all fit into 24 bits (less than 16,777,216).

LED doesn't DIM:

- Measure the port output on the scope or logic analyzer. Make sure the frequency is fixed, but the duty cycle varies.
- Echo the output onto two pins of the microcontroller (output same value to two pins). Your port pin may be damaged.

DAC isn't analog:

- Verify the resistor and capacitor values. Calculate $f=1/(2\pi RC)$, f should be between 1 and 100 Hz.

9.6 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.

- Precision is the number of different PWM outputs that can be generated. This lab describes a system capable of creating about 10,000 different PWMs, which is equivalent to about 13 bits. What could you do to increase the precision?
- What is the relationship between the PWM period (10ms), the resolution of your SysTick timer wait (1us) and the PWM precision? Give an equation for this relationship.
- What would happen in your implementation if you tried to set the PWM period larger than 350ms?
- In what way does the RC circuit model (represent) the behavior of the visual processing of our eyes and brain?
- Why is the RC circuit classified as an analog low pass filter?

  How would you experimentally determine the frequency response of your visual system? One of the early Pokémon anime shows had a 5-sec 12 Hz scene that caused neurological responses in children (search "Pokémon induced seizures").

9.7 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,

- Improve the precision by reducing the units of the timer wait function (e.g., go from 13 bits to 15 bits by reducing timer wait from 1us to 250ns)
- Remove the noise in the PWM DAC (100 Hz ripple in Figure 5), by switching from 100 Hz PWM to 1000 Hz PWM (precision will drop from 13 bits to 10 bits)
- Implement this lab using the hardware timer (we will eventually switch PWM to use the timer)

9.8 Which modules are next?

It turns out the motors using the robot also have a time constant of about 10 Hz. We will use PWM outputs to allow the software to set the power delivered to the motors. However, we cannot use 100% of the processor time to implement the PWMs for our robot. Therefore, we will use hardware timers built into the MSP432 microcontroller, so the software will be free to perform other tasks, while the hardware generates the PWMs automatically. The software will however set the period once, and adjust the duty cycle dynamically to control the behavior of the robot. These are future modules that build on the concepts learned in this module.

Module 10) Use software arrays to verify proper functionality of the system
Module 12) Use this PWM output to adjust power to the DC motor on the robot
Module 13) Use periodic interrupts to create PWM output in hardware

9.9 Things you should have learned

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

- Measure resistance and voltage
- Measure time with a logic analyzer and an oscilloscope
- Create accurate time delays
- Implement PWM output
- Use PWM output to create time-varying behavior
- Create a simple analog low pass filter
- Balance the tradeoffs between range, resolution, and precision

Module 10

Introduction: Debugging Real-time Systems

Introduction: Debugging Real-time Systems

Educational Objectives:

REVIEW C programming arrays
UNDERSTAND how flash memory operates
EXPLORE debugging techniques for real-time systems
LEARN how to generate periodic interrupts using SysTick
INTERFACE bump switches to the robot
DESIGN, BUILD & TEST A SYSTEM
Stores input data into a black-box recorder

Prerequisites (Modules 6, 8, and 9)
- GPIO digital inputs (Module 6)
- Switches and LEDs (Module 8)
- SysTick timer (Module 9)

Recommended reading materials for students:
- Volume 1 Sections 2.7, 6.2, 6.9, 9.1, 9.2, 9.4, and 9.6
  Embedded Systems: Introduction to the MSP432 Microcontroller ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017
- Volume 2 Sections 2.4, 3.9, 5.1, 5.4, and 5.7
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

System verification is an important task when developing embedded systems, especially if the system is to be deployed in safety critical situations. Furthermore, in a **real-time system**, it is not only important to get the correct answer, it is important to get the correct answer at the correct time. **Latency** is the time between when a service is requested and the time when service is initiated. Similarly, **response time** is the time between when a service is requested and the time when service is complete. A real-time system is one that can guarantee a worst-case latency. Alternatively, we can categorize a system as real time if there is an upper bound on the response time.

Some requests occur periodically, and in this module we will use **SysTick interrupts** to execute tasks on a regular basis.

The second component to this module is to develop debugging techniques for real-time systems. **Intrusiveness** is defined as the degree to which the debugging code itself alters the performance of the system being tested. Breakpoints, single stepping, and printf output are high intrusive, and thus not appropriate for debugging real-time systems. Rather we will learn how to dump strategic information into arrays, providing similar observations as the classical printf statement, but in a minimally intrusive manner. For logging, debugging data for long periods of time, we can dump data into the flash ROM of the microcontroller.

In the lab associated with this module, you will interface bump sensors with the microcontroller, see Figure 1. These switches will allow you to know if and where the robot has contacted an obstacle. Data from the line sensor and bump sensors will be collected periodically using SysTick interrupts. Using interrupts to handle the line sensor provides a processor-efficient solution.

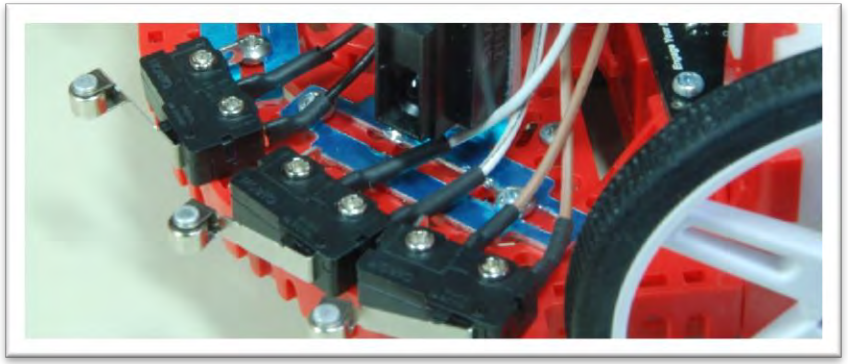


Figure 1. Bump sensors, positioned at the front of the robot.

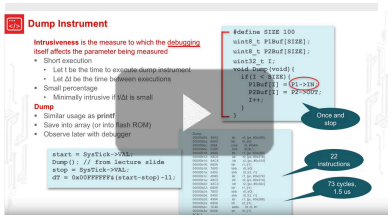
The basic approach to a system requiring multiple software tasks is to deploy multithreading. One software **thread** is the traditional main program, which runs most of the time. Interrupts will be used to create additional threads. An **interrupt** is a hardware-triggered software execution. In this module, the SysTick interrupt will execute a software task periodically. In Module 13, we will use timers to create PWM outputs. In Module 14, we will use edge-triggered interrupts so a software task is executed immediately if any of the bump sensors are activated.

10. TI-RSLK Module 10 – Debugging real-time systems

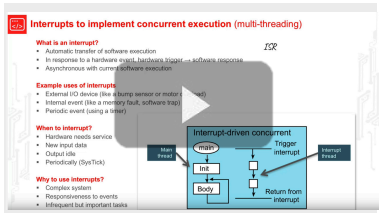
This module provides an intro to how flash memory operates, including debugging techniques for real-time systems and how to generate periodic interrupts using SysTick. Minimally intrusive debugging is essential for real-time systems to evaluate performance while the system runs in real-life situations.

Optionally, download all the curriculum documents for Module 10.

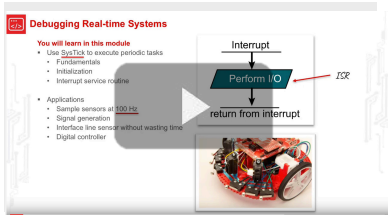
10.1 TI-RSLK Module 10 - Lecture video part I - Debugging real-time systems - Theory
In this module you will learn SysTick timer fundamentals.



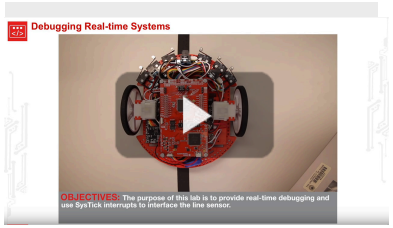
10.2 TI-RSLK Module 10 - Lecture video part II - Debugging real-time systems - Interrupts
You will learn the concept of Pulse Width Modulation (PWM) and duty cycle.



10.3 TI-RSLK Module 10 - Lecture video part III - Debugging real-time systems - SysTick interrupts
The purpose of this lab is to learn about the SysTick timer and use it to create a PWM output.



10.4 TI-RSLK Module 10 - Lab video 10.1 – Demonstrate running the line sensor/black box recorder
The purpose of this lab is to create a PWM output.



Module 10

Lab 10 : Debugging Real-Time Systems

Lab: Debugging Real Time Systems

10.0 Objectives

The purpose of this lab is to interface bump sensors that the robot will use to explore its world, see Figure 1. The line sensor will also be used.

1. You will use arrays to implement minimally intrusive debugging.
2. You will interface bump sensors to the microcontroller.
3. You will learn how to implement SysTick periodic interrupts.
4. You will use interrupts to implement multi-threading.
5. You will implement a black-box recorder by storing data into the flash ROM of the microcontroller.

Good to Know: Interrupts are extremely important for embedded systems, providing a mechanism to implement real-time behavior and multi-threading.

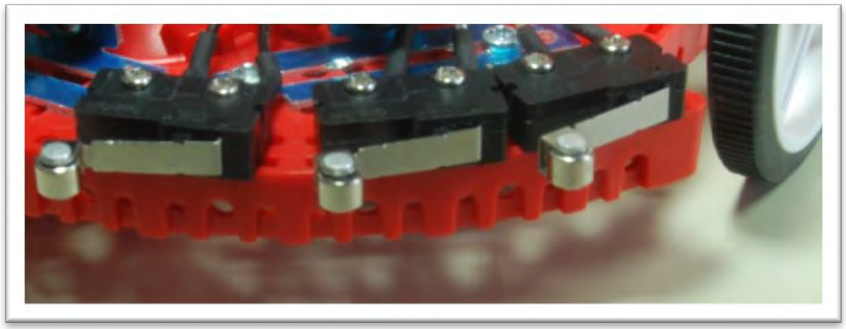


Figure 1. Bump sensors positioned at the front of the robot.

10.1 Getting Started

10.1.1 Software Starter Projects

Look at these three projects:
PeriodicSysTickInt (toggles LEDs using interrupts),
Flash (stores data onto flash ROM),
Lab_Debug (starter project for this lab)

10.1.2 Student Resources

Meet the MSP432 LaunchPad (SLAU596)
MSP432 LaunchPad User's Guide (SLAU597)
QTR-8x.pdf, line sensor datasheet
Polulu_BumpSwitch_1404.png, mechanical drawing of switch

10.1.3 Reading Materials

Volume 1 Sections 2.7, 6.2, 6.9, 9.1, 9.2, 9.4, and 9.6
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 2.4, 3.9, 5.1, 5.4, and 5.7
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

Note: We chose the sampling rate of the sensors to be 100 Hz, because 10 ms is about 10 times shorter than the time constant of the motors used in this robot (100 ms).

10.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
6	Bump switches	Pololu	#1404
1	QTR-8RC Reflectance Sensor Array	Pololu	#961
12	0.5in 2-56 screw	Pololu	2715
12	2-56 nut	MULTICOMP	SPC21805

Table 1. Parts needed for this lab

10.1.5 Lab equipment needed

Oscilloscope (one or two channels at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling.

Lab: Debugging Real Time Systems

.

10.2 System Design Requirements

The first goal of this lab is to implement the line sensor measurement using SysTick interrupts; refer back to Module 6. Similar to Lab 6, you should sample the 8-bit sensor 100 times a second. However, you must perform all input/output with the sensor within executions of the SysTick **interrupt service routine** (ISR). In particular, you must implement the 1-ms delay needed to perform the measurement using subsequent SysTick interrupts. The **priority** of this interrupt doesn't matter in this lab, because it is the only interrupt. However, once we integrate other labs together, this will be a high priority task because measurements need to be real time.

The second goal is to place one to six bump sensors on the robot and interface them to the microcontroller. The bump sensors allow the software to know if and where the robot has collided with an obstacle. Figure 1 shows one possible configuration, and Figure 2 shows a close up of one bump sensor.

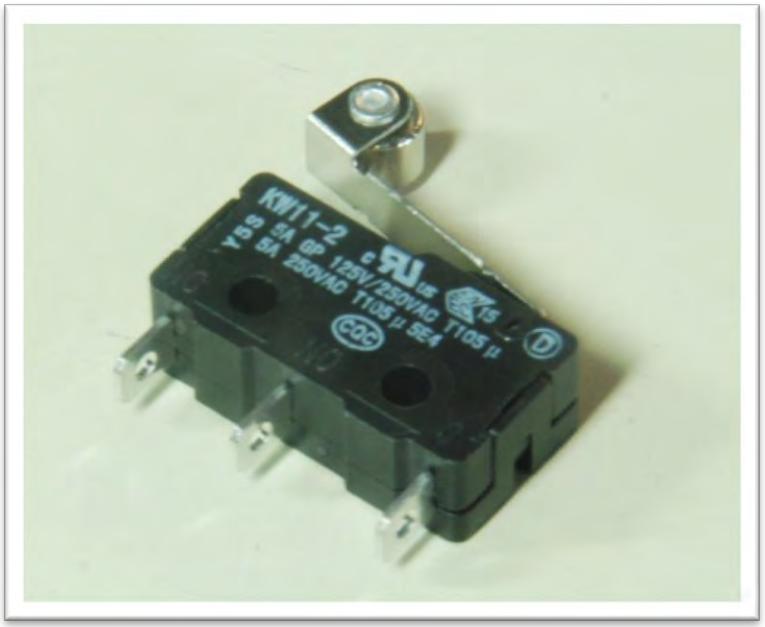


Figure 2. Momentary contact switches allow for collision detection.

The third goal is to develop a minimally intrusive debugging instrument called a **dump**. A dump is a software technique that records strategic information into global arrays. It is classified as minimally intrusive because the time to store data in the arrays (~1µs) is short compared to the time between dumps (100ms). During robot operation, software writes into the arrays, and when the experiment is complete, you use the debugger to observe values.

You will write two debugging functions to implement the dump.

a) **Debug_Init** initializes the array(s).

b) The function **Debug_Dump** takes two parameters and saves the data into RAM. There are two 8-bit parameters to record (8-bit bump sensor data and 8-bit line sensor data). Your system should allow a least 256 measurements (512 bytes).
Your SysTick ISR will call **Debug_Dump** 100 times per second, whenever a new measurement is complete. Since the arrays have 256 entries, this RAM-based recording allows up to 2.56 seconds of debugging. Once the arrays are full, the dump pointers (or indices) will wrap back to the beginning and overwrite the oldest data. In this way, at any given time, the last 2.56 seconds of sensor data are available in the arrays.

The fourth goal is to extend the dump operation to continuously store sensor data onto the **flash ROM** of the microcontroller, implementing the black box recorder. The flash ROM is larger than the RAM, but it too is finite. Therefore eventually, the ROM space will fill. More specifically you will use 128 kibibytes of flash ROM (addresses 0x00020000 to 0x0003FFFF). These addresses are in Flash Bank1. Your program will reside in Flash Bank 0 (addresses 0x00000000 to 0x0001FFFF), allowing you to execute code at the same time as you write to ROM. If you record 200 bytes/sec, you can save data for 655 seconds, or almost 11 minutes.

You will implement two more debugging functions

a) **Debug_FlashInit() which** will erase the 128 kibibytes of flash ROM, addresses 0x00020000 to 0x0003FFFF. Erasing ROM sets the data to 0xFF. You may pick any block size from 32 bytes to 512 bytes. Let 2^n be your block size. There are $2^{17}/2^n$ blocks in this 128k space. If the data of a block are 0xFF, then the block is considered empty.

b) The function **Debug_FlashRecord()** will record 2^n bytes into the next free block on the flash ROM. You will be able to observe the recorded data later using the debugger.

Lab: Debugging Real Time Systems

Note: The line sensor measurements and RAM recording will occur in the SysTick ISR, but all flash operations must occur from the main program. The line sensor generates an 8-bit number and the bump sensors generate an 8-bit number. At a sampling rate of 100 Hz, the system generates 200 bytes/sec. Therefore, every $0.05*2^n$ seconds you will write 2^n bytes to flash ROM. When the ROM is full after 655 seconds, you should stop the ROM recording.

10.3 Experiment set-up

You learned how to interface the QTR-8RC line sensor back in Lab of Module 6. You also learned how to interface switches in Lab 8. In this lab you will attach bump switches to the front of the robot and interface them to the LaunchPad. You can use 2-56 screws and nuts to position the bump sensors on the edge of the front of the robot. Figure 3 shows one possible placement for six sensors.

Warning: *TI MSP432 pins are not 5V tolerant; you must power the line sensor and bump sensors with +3.3V.*

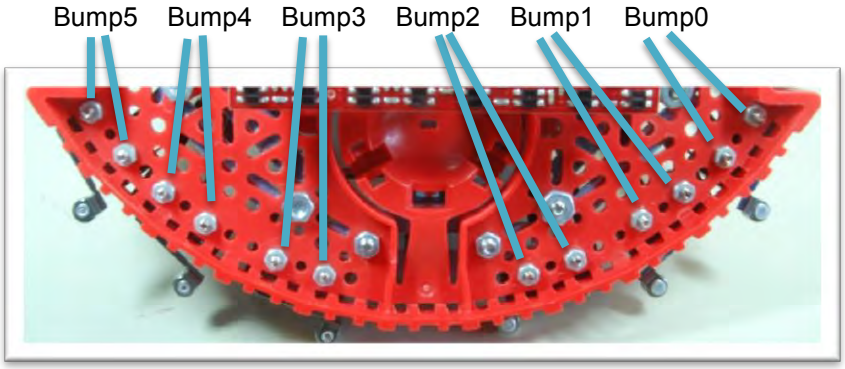


Figure 3. Bump sensors attached to the front of the robot (bottom view).

10.4 System Development Plan

10.4.1 Using SysTick interrupts to read the line sensor

You will perform the line sensor input in the background using SysTick interrupts. If the SysTick interrupts are occurring at 1000 Hz (every 1ms), then in one execution of the ISR you

1. Set P5.3 high (turn on 8 IR LED)
2. Make the P7.7-P7.0 outputs, and set them all high
3. Wait 10 µs
4. Make the P7.7-P7.0 inputs

You will define the above four steps as the function **Reflectance_Start()**, and call this function every tenth time in the SysTick ISR. The second part of the measurement occurs in the subsequent ISR, where you

5. Read the 8-bit sensor result
6. Turn off the 8 IR LEDs (P5.3) low
7. Store the data into a shared global variable

You will define steps 5 and 6 as the function **uint8_t Reflectance_End()**. Step 8 will occur in the SysTick ISR itself. If you are sampling the line sensor at 100 Hz, there will be eight SysTick interrupts during which the software performs no operation, one interrupt that calls **Reflectance_Start()**, and one interrupt that calls **Reflectance_End()**.

It is good debugging style to toggle a port pin during each ISR execution. Place the sensor data in a memory watch window and use the debugger, and oscilloscope to verify the sensor behaves in a similar manner to Lab 6.

Lab: Debugging Real Time Systems

10.4.2 Interfacing the bump sensors

You can implement positive logic or negative logic. You can implement internal resistors or external resistors.

Note: If you are going to do the Wifi lab, we recommend using pins P8.7-P8.3, P8.0 for the six bump sensors, because these pins do not conflict with the Wifi lab. If you are going to do the edge-triggered interrupt lab, we recommend using pins P4.7-P4.2 for the six bump sensors, because these pins can cause edge-triggered interrupts.

You will write one function to initialize the bump sensors, **Bump_Init()**. This function simply sets the appropriate port pins and enables internal resistors as needed. A second function, **uint8_t Bump_Read()**, reads the switches and returns one 8-bit result. Every time the SysTick ISR performs a call to **Reflectance_End()**, you should also call **Bump_Read()**, and you should place the result in a shared global variable and set a semaphore.

Note: In subsequent labs when the robot is moving, you will handle collisions within this SysTick ISR after calling **Bump_Read()**. The worst case latency of a collision event will therefore be 10 ms.

10.4.3 Debugging dump

There are a number of design choices for implementing the debugging dump. You could create two 8-bit arrays, one for the line sensor and one for the bump sensor. Alternatively, you could create one 16-bit array and pack the two sensor readings into one 16-bit data value. The specifications call for at least 256 recordings, but you could increase this value if you wish. The MSP432 microcontroller has 64 kibibytes of RAM, and this debugging feature should only use a small fraction of available RAM. Another choice is whether to use a pointer or an index to access the array. This typically involves a tradeoff between execution speed and software style. You could implement both and observe the assembly code generated by each version. To reduce intrusiveness, we suggest you choose the method that executes the fastest (i.e., the fewest assembly instructions.)

One way to quantitatively measure the intrusiveness of the debugging instrument is to count the number of assembly instruction it takes to implement **Debug_Dump()**. A better method is to use an oscilloscope and an unused port pin, as illustrated in Program10_1.

```
int Program10_1(void){ uint8_t data=0;
  Clock_Init48MHz();
  Debug_Init();
  LaunchPad_Init();
  while(1){
    P1->OUT |= 0x01;
    Debug_Dump(data,data+1);// linear sequence
    P1->OUT &= ~0x01;
    data=data+2;
  }
}
```

10.4.4 Black box recorder

When debugging the black box recorder, we recommend you begin by using simple main programs, like Program10_2 and Program 10_3. This way you can test your two functions separately. Use the debugger to verify ROM is correctly programmed.

```
// Driver test
#define SIZE 256  // feel free to adjust the size
uint16_t Buffer[SIZE];
int Program10_2(void){ uint16_t i;
  Clock_Init48MHz();
  LaunchPad_Init(); // built-in switches and LEDs
  for(i=0;i<SIZE;i++){
    Buffer[i] = (i<<8)+(255-i); // test data
  }
  i = 0;
  while(1){
    P1->OUT |= 0x01;
    Debug_FlashInit();
    P1->OUT &= ~0x01;
    P2->OUT |= 0x01;
    Debug_FlashRecord(Buffer);
    P2->OUT &= ~0x01;
    i++;
  }
}
```

```
int Program10_3(void){ uint16_t i;
  Clock_Init48MHz();
  LaunchPad_Init(); // built-in switches and LEDs
  for(i=0;i<SIZE;i++){
    Buffer[i] = (i<<8)+(255-i); // test data
  }
  P1->OUT |= 0x01;
  Debug_FlashInit();
  P1->OUT &= ~0x01;
  i = 0;
  while(1){
    P2->OUT |= 0x01;
    Debug_FlashRecord(Buffer);
    P2->OUT &= ~0x01;
    i++;
  }
}
```

Use an oscilloscope or logic analyzer to measure the execution time of the functions. Calculate the maximum data rate achieved by the ROM programming (number of bytes in the block divided by the time to write the block). This data rate will be much faster than the 200 byte/sec data production rate. However, depending on the size of the buffer, it may or may not take more than 10 ms to program one buffer.

When integrating the black box recorder with the sensor measurements, use a shared global flag (**Semaphore**) that is set in the SysTick ISR when the buffer is full. You will read the flag in the main program to know when to make the next call to **Debug_FlashRecord()**. One way to make sure the system is properly recording all the data in the correct order is to measure actual line and bump sensor input, but record test data as generated in Program10_2. Using test data allows you to see if any points are lost or duplicated. Your final main program will have this sort of behavior.

```
while(1){
  if(Semaphore==1){ // wait for flag to be set
    P2->OUT |= 0x01;
    Debug_FlashRecord(Buffer);
    P2->OUT &= ~0x01;
    Semaphore = 0; // clear flag
  }
}
```

10.5 Troubleshooting

Bump sensors don't work:

- Check the wiring as described in Module 8
- Look at signals with a voltmeter, scope or logic analyzer
- Look at the port registers in the debugger

Flash storage doesn't work:

- Run the *Flash project code* to test the hardware
- Observe *the Flash project* to see how to call the low-level driver

10.6 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.

- In this lab we just stored data, but not the time at which the data was sampled. In this application why did we not save time as well?
- What does it mean when we classify the line sensor and bump sensor interfaces as real time? Why is it important for these inputs to be real time?
- Why should we use internal resistors instead of external resistors for the bump interfaces?
- We specified the latency to be a maximum of 10 ms. What is the average latency? How could we have redesigned this to reduce latency?
- Why is it good design to have the IR LED off for 90% and on for only 10% of the time?
- What happens when we erase ROM? What happens if a ROM bit is already 0, and we try to program it to 1?
- Assume we call **Debug_FlashInit()** once when the robot is manufactured, but subsequent runs of the main program do not erase the ROM. What will be recorded in ROM? I.e., what happens to this data if we remove and later restore power?

10.7 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,

- One of the techniques to increase the recording time is to only store data when it changes. However, in this scheme you do need to record data and the time the data changes. A global variable incremented in the SysTick ISR could hold the system time in ms.
- A **critical section** is software behavior that varies depending on relative execution of two seemingly unrelated pieces of code. For example executing P2->OUT^=0x01 in the ISR and then executing P2->OUT ^=0x02 in the main program will create a bug due to the critical section. Sharing a common port in two different threads generates a critical section. You could rewrite the LaunchPad driver code (LaunchPad.c) to use bit-banding, which will remove some critical sections.
- You could include **UART0.c** in the project and create an additional debugging function that dumps the data that was recorded in flash ROM out to the serial port. You can then run a program like PuTTY or TExasDisplay to see the data. This feature will allow you to capture data on the PC for report writing and documentation.

10.8 Which modules are next?

This was our first of many uses of interrupts in this course. The following modules will build on this module:

Module 12) Connect the motors to the robot.
Module 13) Use timers to create PWM signals, and use interrupts to manage multiple software tasks.
Module 14) Use edge-triggered interrupts to a software task immediately upon a switch contact.

10.9 Things you should have learned

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

- Use interrupts to implement multithreading
- Use global variables to communicate between threads
- Optimize execution speed when accessing arrays
- Create minimally intrusive debugging tools
- Perform execution profiling using port pins and a scope
- Erase and program flash ROM for logging debugging data

Module 11

Introduction: Liquid Crystal Display

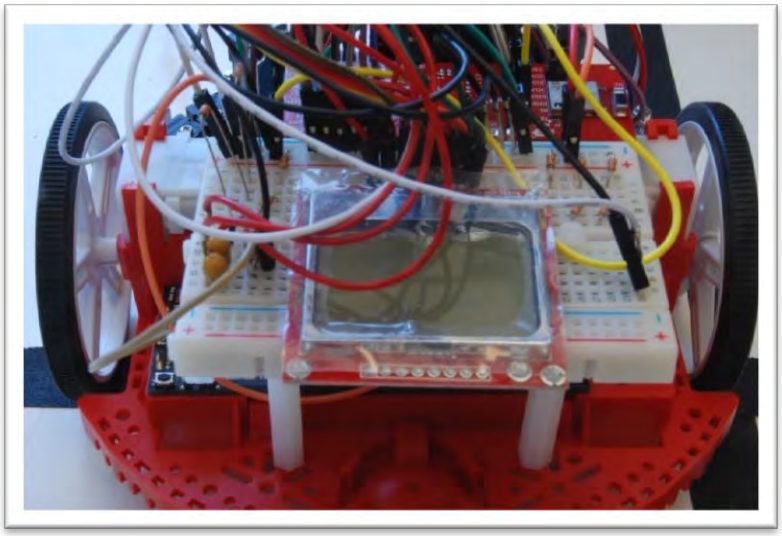
Introduction: Liquid Crystal Display

Educational Objectives:

REVIEW Software/hardware synchronization with busy-wait
UNDERSTAND synchronous serial communication
DEVELOP a set of display output functions
LEARN how to display characters on an LCD screen
DESIGN, BUILD & TEST A SYSTEM
Interface an LCD to the microcontroller

Prerequisites (Modules 1, 4, and 6)
- Running code on the LaunchPad using CCS (Module 1)
- Basic C programming (Module 4)
- GPIO (Module 6)

Recommended reading materials for students:
- Volume 1 Sections 4.5, 7.6, 7.7, 8.3, and 8.4
  Embedded Systems: Introduction to the MSP432 Microcontroller ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017
- Volume 2 Sections 1.5, 3.4, and 7.5
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017



Microcontrollers employ multiple approaches to communicate synchronously with peripheral devices and other microcontrollers. The synchronous peripheral interface (SPI) system can operate as a master or as a slave. The channel can have one master and one slave, or it can have one master and multiple slaves. In this module, the MSP432 will be the master and the LCD will be the slave. The master initiates all data communication.

A universal asynchronous receiver transmitter (UART, see Module 18) implements an asynchronous protocol, meaning the data are transmitted without timing information and the receiver derives time from the data. SPI implements a synchronous protocol, meaning transmitter and receiver operate off the same clock. Two devices communicating with asynchronous serial interfaces operate at the same frequency (baud rate) but have two separate clocks. With a UART protocol, the clock signal is not included in the interface cable between devices. Two devices communicating with SPI operate from the same clock (synchronized). With an SPI protocol, the clock signal is included in the interface cable between devices. Typically, the master device creates the clock, and the slave device(s) uses the clock to latch the data (in or out.)

The SPI protocol includes four I/O lines. The slave select STE is an optional negative logic control signal from master to slave signal signifying that the channel is active. The second line, CLK, is a 50% duty cycle clock generated by the master. The slave in master out (SIMO) is a data line driven by the master and received by the slave. The slave out master in (SOMI) is a data line driven by the slave and received by the master. In order to work properly, the transmitting device uses one edge of the clock to change its output, and the receiving device uses the other edge to accept the data.

In the lab associated with this module, we will interface a Nokia 5110 LCD using busy-wait synchronization. Before we output data or commands to the display, we will check a status flag and wait for the previous operation to complete. Busy-wait synchronization is very simple and is appropriate for I/O devices that are fast and predicable. Debugging is a critical task when developing complex systems. The motivation for this lab is to provide a real-time monitoring, so you can visualize debugging parameters as the robot is exploring its world.

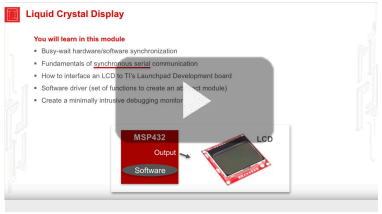
11. TI-RSLK Module 11 – Liquid Crystal Display

This module will show you how to display characters and provide real-time debugging on an LCD screen. An LCD on your robot provides a convenient way to observe what it is thinking.

Optionally, download all the curriculum documents for Module 11.

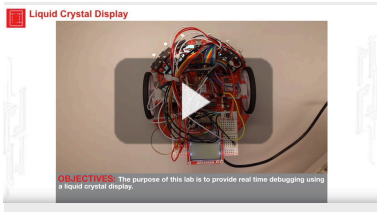
11.1 TI-RSLK Module 11 - Lecture video - Liquid Crystal Display

In this module you will learn how to interface a LCD to TI's LaunchPad development kit.



11.2 TI-RSLK Module 11 - Lab video 11.1 – Demonstrate LCD interface

Review Software/hardware synchronization with busy-wait and understand synchronous serial communication.



Module 11

Lab 11: Liquid Crystal Display

Lab: Liquid Crystal Display

11.0 Objectives

The purpose of this lab is to interface an LCD display to the microcontroller and develop a set of software routines to assist in debugging subsequent labs on the robot.

1. You will connect an LCD to the microcontroller.
2. You will use the synchronous serial protocol to communicate.
3. You will implement a set of software functions for the LCD.

Good to Know: Even though the LCD doesn't actually contribute to the input-calculate-output loop required to solve the robot challenge, it will be extremely useful when debugging when the robot is running untethered in the arena.

11.1 Getting Started

11.1.1 Software Starter Projects

Look at this project:
Lab11_FSM (starter project for this lab)

11.1.2 Student Resources (in datasheets directory)

Nokia5110.pdf (data sheet for the LCD)

11.1.3 Reading Materials

Volume 1 Sections 4.5, 7.6, 7.7, 8.3, and 8.4
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 1.5, 3.4, and 7.5
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller",

Figure 1. Bump sensors positioned at the front of the robot.

11.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	LCD display	Nokia	5110

11.1.5 Lab equipment needed

Oscilloscope (two channels with at least 10MHz sampling)
Or Logic Analyzer (4 channels with at least 10MHz sampling)

11.2 System Design Requirements

The overall goal of this lab is to interface an LCD to the microcontroller and use it to output characters and numbers. The Nokia 5110 is a monochrome display that is 84 pixels wide by 48 pixels high. Each character is defined by a 5 pixels wide by 8 pixels high image. You can see the font table as a constant array called **ASCII** inside **Nokia.c** starter file. For example, the letter '7' is defined in line 132 as

```
{0x01, 0x71, 0x09, 0x05, 0x03}
```

This 40-bit value creates the image shown in Figure 1 on the display. The 0x01 is the first column and 0x03 is the last column, with bit 0 on top.

Each character has bit 7 clear to make a space between lines. The function **Nokia5110_OutChar** will place a blank vertical line in front of and one blank line after each character. This means each character requires a 7 by 8 pixel area to print. Since the display is 84 wide by 48 high, this font size allows for 84/7= 12 characters on each line, and allows 48/8=6 lines.

Lab: Liquid Crystal Display

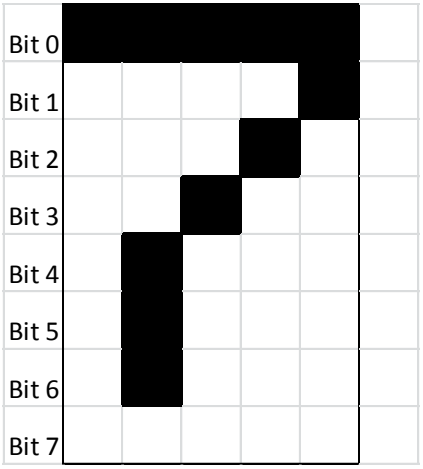


Figure 1. Pixel image to create the number 7, showing a blank vertical line before and after the character. Bit 7 is clear for all characters.

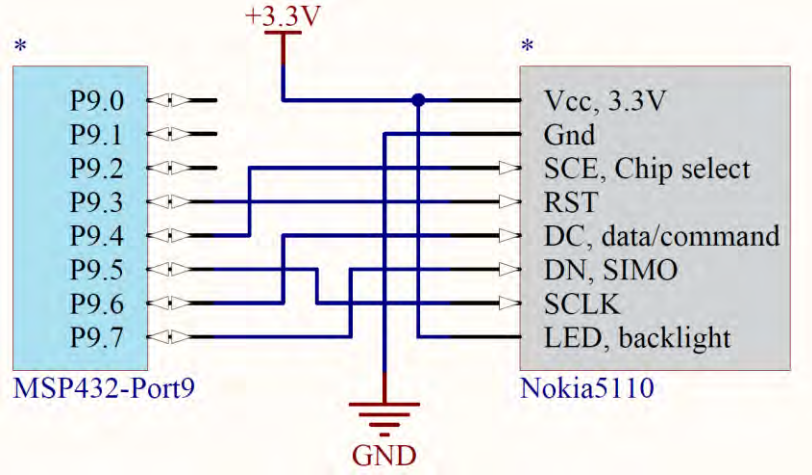
On the low level, you are required to write one routine that sends an 8-bit command to the LCD (**lcdcommandwrite**) and a second routine that sends 8-bit data to the LCD (**lcddatawrite**). These routines use busy-wait synchronization with the SPI synchronous serial interface. These are private functions, so prototypes are not available in the **Nokia5110.h** header file.

On the high level, you are required to write one routine that outputs a string (**Nokia5110_OutString**) and a second routine to output an unsigned 16-bit decimal number (**Nokia5110_OutUDec**).

You will find the **Nokia5110.h** and **Nokia5110.c** files in the **inc** folder. This means after you complete this lab, you can use these functions in the remaining labs.

11.3 Experiment set-up

You will implement this lab using the MSP432 LaunchPad and the LCD. Figure 2 shows Port 9 can be used to interface the LCD. However you may use any pins that support the synchronous serial port (SPI protocol). Pins P9.7, P9.5, and P9.4 are configured for SPI mode. Pins P9.3 and P9.6 are set to be regular digital outputs. DC=1 means data, and DC=0 means command. The reset pin is used to initialize the LCD hardware (RESET=0, at least 100ns wait, RESET=1).



11.4 System Development Plan

11.4.1 lcdcommandwrite

To send one 8-bit command, your **lcdcommandwrite** function should perform the following four steps, even though 1) and 4) are the same.

1) Wait for the SPI to be idle (let previous frame finish), **UCBUSY**
2) Set DC for command (0)
3) Write 8-bit command to the SPI data register (**TXBUF**), starts SPI
4) Wait for the SPI to be idle (let this transmission finish), **UCBUSY**

Look up in the Nokia5110 and MSP432 data sheets what the expected waveforms should look like when **lcdcommandwrite** is called in the program. Use an oscilloscope or logic analyzer to verify the waveforms are as expected. You can use a simple program like the following to test this low-level function.

```
void Testlcdcommandwrite(void){
  while(1){
    lcdcommandwrite(0x21);
  }
}
```

11.4.2 lcddatawrite

To send one 8-bit data, your **lcddatawrite** function should perform the following three steps. Skipping step 4) when outputting data makes it run much faster.

1) Wait for the SPI to be idle (let previous frame finish), **UCBUSY**
2) Set DC for data (1)
3) Write 8-bit data to the SPI data register (**TXBUF**), starts SPI

11.4.3 Nokia5110_OutString

There is a midlevel function given to you called **Nokia5110_OutChar** that outputs one character to the LCD. You will use this function to implement a function called **Nokia5110_OutString** that outputs a string to the LCD.

11.4.4 Nokia5110_OutUDec

Similarly, you will use **Nokia5110_OutChar** to implement a function called **Nokia5110_OutUDec** that outputs a 16-bit unsigned number to the LCD. One of the specifications for this function is that the image is created right justified that fills exactly 5 characters. This functionality allows you to output numbers on the LCD that are easy to read.

More specifically, for numbers 0 to 9, you will output 4 spaces and the one digit. For numbers 10 to 99, you will output 3 spaces followed by two digits. For numbers 100 to 999, you will output 2 spaces followed by three digits. For numbers 1000 to 9999, you will output 1 space followed by four digits. For numbers 10000 to 65535, you will output all five digits.

To illustrate how this function could be used, consider the example where the LCD contains debug data continuously updated during a robot run, You could execute this code once at the beginning

```
Nokia5110_SetCursor(0,2);  // left, third row
Nokia5110_OutString("D= ");
Nokia5110_OutUDec(0);
Nokia5110_OutString(" mm");
```

Then, when you wish to update the LCD with a new distance value, you can just output the 5 characters of the new value. This method will make a pretty display that will not flicker as the numbers change.

```
Nokia5110_SetCursor(3,2);  // spot for number
Nokia5110_OutUDec(myDistance);
```

Use a debugging profile to measure how long it takes to execute **Nokia5110_OutUDec.** Knowing that the 12 MHz SPI clock is 12 MHz, explain why this measurement is reasonable.

11.5 Troubleshooting

The LCD does not display characters:

- Check all the connections between LaunchPad and LCD.
- Make sure RESET is high.
- Run a simple main program that calls **lcdcommandwrite** over and over with the same command. Use a logic analyzer or scope to verify all five signals from LaunchPad to LCD are proper. Section 12.1 of the data sheet describes the expected SPI protocol.

Back light does not operate:

- Check the ground.
- Verify you have pin 1 properly identified and haven't wired it backwards.

11.6 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.

- What does it mean that this interface is serial? Why is serial important?
- What does it mean that this interface is synchronous? Why is synchronous important?
- What is the purpose of each of these three files: **Nokia5110.h**, **Nokia5110.c**, and **Lab11_LCDmain.c**?
- How long does it take to execute **Nokia5110_OutUDec**? Is it appropriate to call this function during an ISR, or is it better to always perform LCD output in the main program?

Lab: Liquid Crystal Display

11.7 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,

- Add an output function for 16-bit signed numbers.
- Add an output function for signed 32-bit numbers
- Interface a different LCD, such as the ST7735R
- Create a set of functions that plots data versus time on the LCD

11.8 Which modules are next?

Modules 1-11 have introduced the basics of the microcontroller. The next set of modules allow for more complex functionality for the robot.

Module 12) interface the motors to the robot.
Module 13) write software to adjust power to the motors.
Module 14) use interrupts to detect collisions in real time
Module 15) interface IR distance sensors to measure distance to the wall.
Module 16) interface tachometers to measure wheel speed.

11.9 Things you should have learned

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

- Understand busy-wait and know why busy-wait was used for this interface and not interrupts
- Synchronous serial protocol: how it works and why it is important
- The concept of minimally intrusive debugging

Module 12

Introduction: DC motors

Introduction: DC motors

Educational Objectives:

UNDERSTAND Electromagnetic and mechanical model of a DC motor
INTERFACE The circuits needed to drive power to the DC motors
EXTEND The PWM software to provide software control of the motors
MEASURE Motor speed versus duty cycle

Prerequisites (Modules 2, 5, 9)
- Voltage, current and power (Module 2)
- Resistance, capacitance (Module 2)
- Battery and Voltage regulation (Module 5)
- SysTick Timer (Module 9)

Recommended reading materials for students:
- Volume 1 Sections 8.1, 8.6, and 8.7
  Embedded Systems: Introduction to the MSP432 Microcontroller, ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or
- Volume 2 Sections 1.4 and 6.5
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

This module, together with the next (Module 13), will develop the robot so it moves, see Figure 1. Back in Module 9 you created software using *pulse width modulation* that dimmed an LED. You will now use that PWM software to adjust the power delivered to the DC motors on the robot.

The focus of this lab is the mechanical and electrical aspects of the motors, using **two H-bridges**. This is common circuit build to spin motors. You will also learn about a motor driver IC TI- DRV8838 which is used to interface the two DC motors to the microcontroller. Module 13 will focus on using the timers to create flexible and efficient software for generating two PWM outputs that will provide power to your DC motor.

In this lab, you will measure motor speed with your eyes and a stopwatch. However, in Module 16, you will interface a tachometer so the software can measure motor speed directly. Then, you will combine Modules 12, 13, 16, and 17 to create a **closed-loop control system** that allows you to set the desired speed of each motor.

The electrical **power** (P in watts) delivered to the motor is the product of voltage (V in volts), current (I in amps), and duty cycle (Duty as a dimensionless fraction 0 to 1, studied in Module 9). Motors can spin forward or backward because voltage and current have direction or polarity.

$$P = V * I * Duty$$

On this robot, voltage will be fixed at about 7V, and current will depend on the mechanical load (friction). Software, however, will set the duty cycle. The motor converts electrical power into mechanical power. This mechanical power delivers **torque** to the wheel (torque = force*distance), causing the wheel to spin and the robot to move.

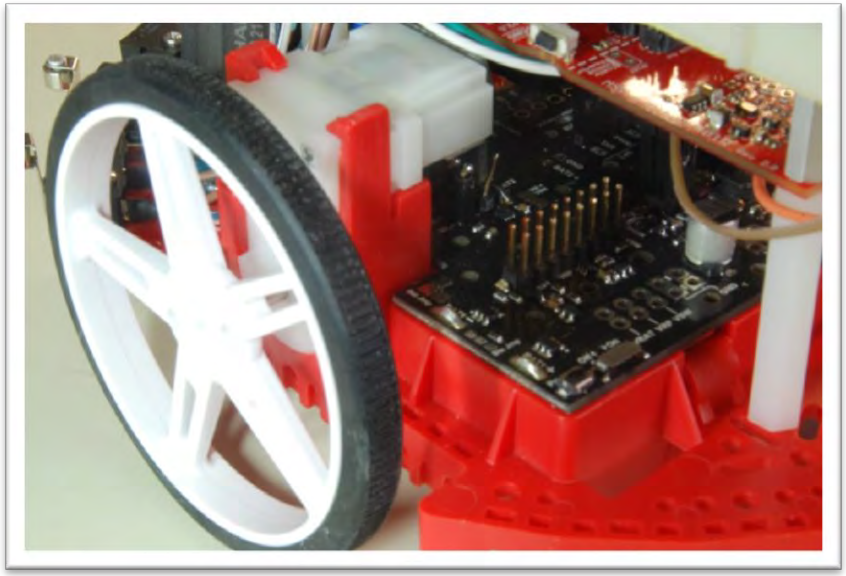


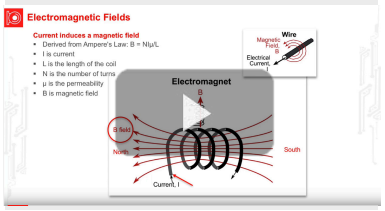
Figure 1. In this lab you add motors and wheels to the robot.

12. TI-RSLK Module 12 – DC motors

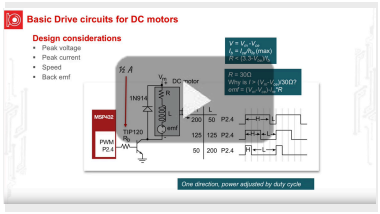
The purpose of this lab is to interface the motors to the TI LaunchPad to make the robot move. Understanding how duty cycle, voltage and current combine to affect speed is required when building your robot.

Optionally, download all the curriculum documents for Module 12.

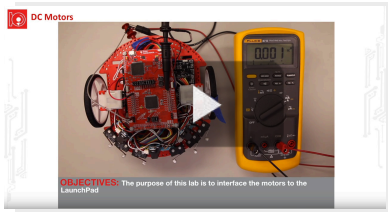
12.1 TI-RSLK Module 12 - Lecture video part I - DC motors - Physics
In this module you will receive an overview of the circuits needed to drive power to the DC motors.



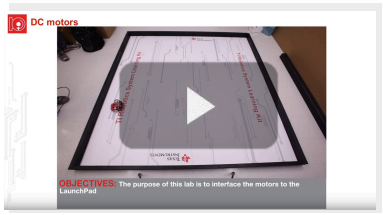
12.2 TI-RSLK Module 12 - Lecture video part II - DC motors - Interface
The focus of this module is the mechanical and electrical aspects of the motors.



12.3 TI-RSLK Module 12 - Lab video 12.1 – Demonstrate motor fundamentals
The purpose of this lab is to interface the motors to the LaunchPad.



12.4 TI-RSLK Module 12 - Lab video 12.2 – Demonstrate robot moving in a preset pattern
The goal of this lab is to see how straight the robot moves if we were to set the two duty cycles to an equal value.



Module 12

Lab 12: DC motors

Lab: DC motors

12.0 Objectives

The purpose of this lab is to build the electronics needed to spin the motors. The hardware interface includes an H-bridge motor driver using the TI DRV8838 driver that allows the software to spin each motor forward or backward. The software can vary the **electrical power** delivered to each motor using **pulse width modulation** (PWM). In this module,

1. You will learn the electromagnetic aspects of the motor.
2. You will attach the motors and wheels to the robot.
3. You will use the driver board to interface the motors to the microcontroller.
4. You will measure the voltage and current to the motors.
5. You will perform an analysis of the behavior of the motor, plotting motor speed versus duty cycle.

Good to Know: Even though you will measure motor speed as a function of duty cycle, this relationship depends on many factors that can change over time, such as motor efficiency, battery voltage, voltage drop in the H-bridge, mechanical forces, and friction. For all practical purposes, without sensors, the software can only choose to go faster or to go slower, but it cannot set the motor speed. On this robot, there are two motors in differential drive configuration. This means even the simplest operation such as moving in a straight line will require sensor feedback. There are three such sensors available in this course: the line sensor (Module 6), the IR distance sensors (Module 15), and the tachometer (Module 16).

12.1 Getting Started

12.1.1 Software Starter Projects

Look at these two projects:
Lab09_SysTick (your solution to Lab 9)
Lab12_Motors (starter project for this lab)

Note: Please do not use the voltmeter, oscilloscope or logic analyzer created by TExaS for this lab. Voltages applied to inputs of the MSP432 must remain between 0 and 3.3V. Voltages outside this range will damage the MSP432.

12.1.2 Student Resources (in datasheet directory)

MotorDriverPowerDistribution.pdf	Data sheet for power board
Pololu Romi Chassis User's Guide.pdf	How to build the robot
drv8838.pdf	Data sheet for the H-bridge driver

12.1.3 Reading Materials

Volume 1 Sections 8.1, 8.6, and 8.7
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 1.4 and 6.5
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

Read the specifications on the Motor Driver and Power Distribution board
website https://www.pololu.com/product/3543
https://www.pololu.com/docs/0J68

12.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	Romi Chassis Kit - Red	Pololu	3502
1	Motor Driver and Power Distribution Board for Romi	Pololu	3543
1	Romi Encoder Pair Kit, 12 CPR* (optional)	Pololu	3542
2	Rechargeable Battery, Pack of 4, Metal Hydride 1300 mAh, 1.2V, AA	Energizer	626831
4	1.375in 4-40 Nylon standoff	Keystone	4809
2	0.187in 4-40 metal nut	Keystone	4694
6	0.5in 4-40 Nylon machine screw	Pololu	1962

12.1.5 Lab equipment needed

Oscilloscope (one or two channels at least 10 kHz sampling)

Lab: DC motors

Voltmeter, ohmmeter, and current meter

12.2 System Design Requirements

The goal of this lab is to place the motors and wheels on the robot and configure the motor control board so the software can control the two motors. The Motor Driver and Power Distribution Board (**MDPDB**) used in Module 5 lab also includes two H-bridge drivers (TI DRV8838) that provide the voltage and current needed to spin the motors.

First, you will mechanically build, and then electrically connect the two motors, two wheels, the caster, and the **MDPDB**. Six control signals will be connected from the microcontroller to the **MDPDB** so the software can control both motors {forward, stop, reverse}. Furthermore, you will use the PWM software from Lab 9 to adjust the delivered power to the two wheels.

The second part of this lab is to study the behavior of the motor. You will measure voltage (volts), current (amps), average power (watts), and rotational speed (rpm) of the DC motor as a function of duty cycle.

The outcome of this lab is to build a system that drives in more or less a straight line until one of the bump sensors detects a collision.

12.3 Experiment set-up

The first step is to read the data sheet for the Romi chassis, and follow the directions on **https://www.pololu.com/docs/0J68/all** to connect the two wheels, caster, two motors, and motor board per instructions to the Romi chassis. Figure 1 shows some of the parts needed for the robot.

Note A: If you do not intend to buy and build the tachometer, labeled as Encoder in Figure 1 (used in Lab 16 with the Romi Encoder Pair Kit, 12 CPR https://www.pololu.com/product/3542), then you will solder four wires from the two motors to the motor board (MR+, MR-, ML+, ML-).

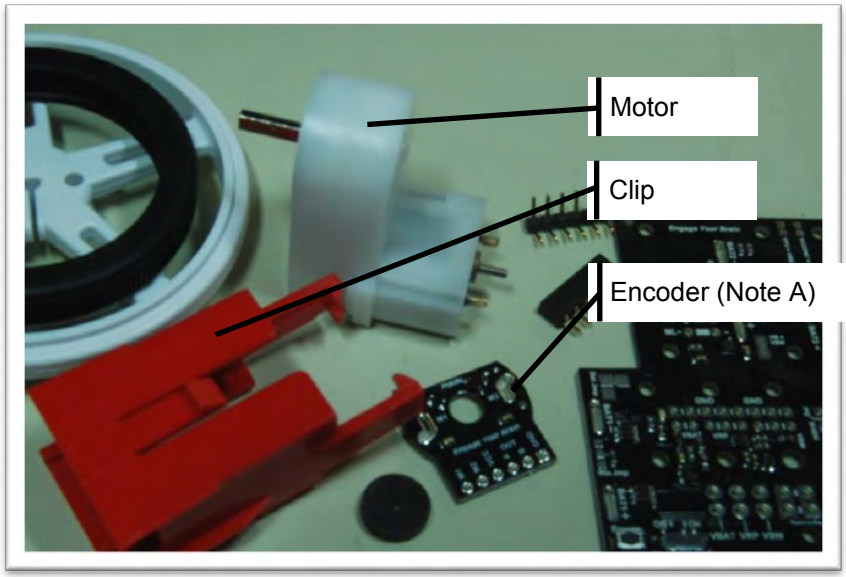


Figure 1. Parts needed to build the motor system.

Next, you will connect six wires from the **MDPDB** to the LaunchPad. Since these signals are on the regular LaunchPad connectors, you can use either male or female wires on the LaunchPad side (the robots in the figures use female connectors). Figure 2 shows a possible interface circuit. On the **MDPDB** side you can solder wires directly, or solder a male header into the **MDPDB** and use female-female cables, see Figure 3. Refer to the data sheet of the DRV8838 to see how the software output values to these six signals affect motor behavior.

Lab: DC motors

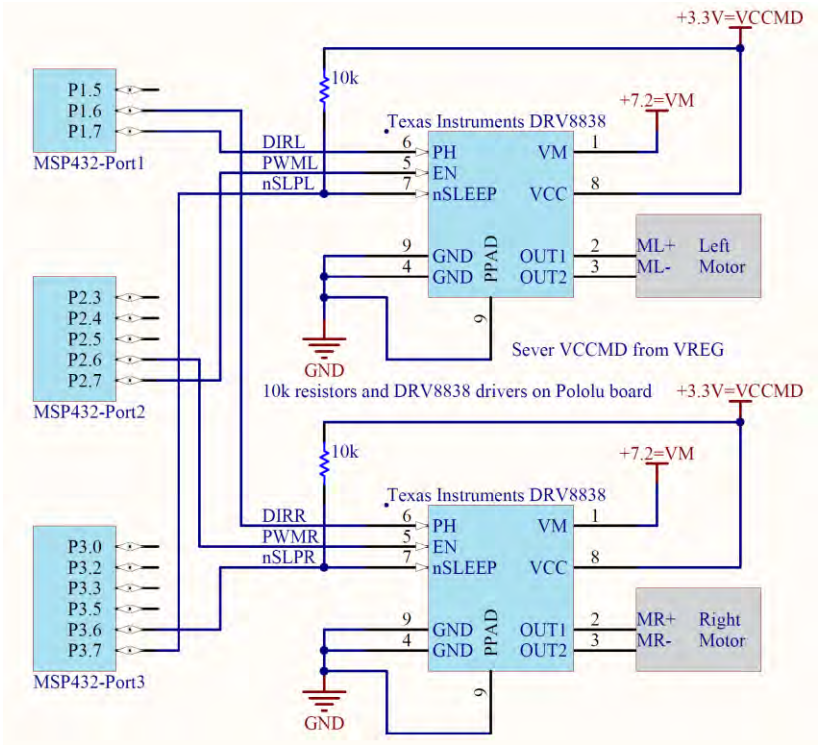


Figure 2. Interface circuit.

LaunchPad	MDPDB	DRV8838	Description
P1.6	DIRR	PH	Right Motor Direction
P3.6	nSLPR	nSLEEP	Right Motor Sleep
P2.6	PWMR	EN	Right Motor PWM
P1.7	DIRL	PH	Left Motor Direction
P3.7	nSLPL	nSLEEP	Left Motor Sleep
P2.7	PWML	EN	Left Motor PWM

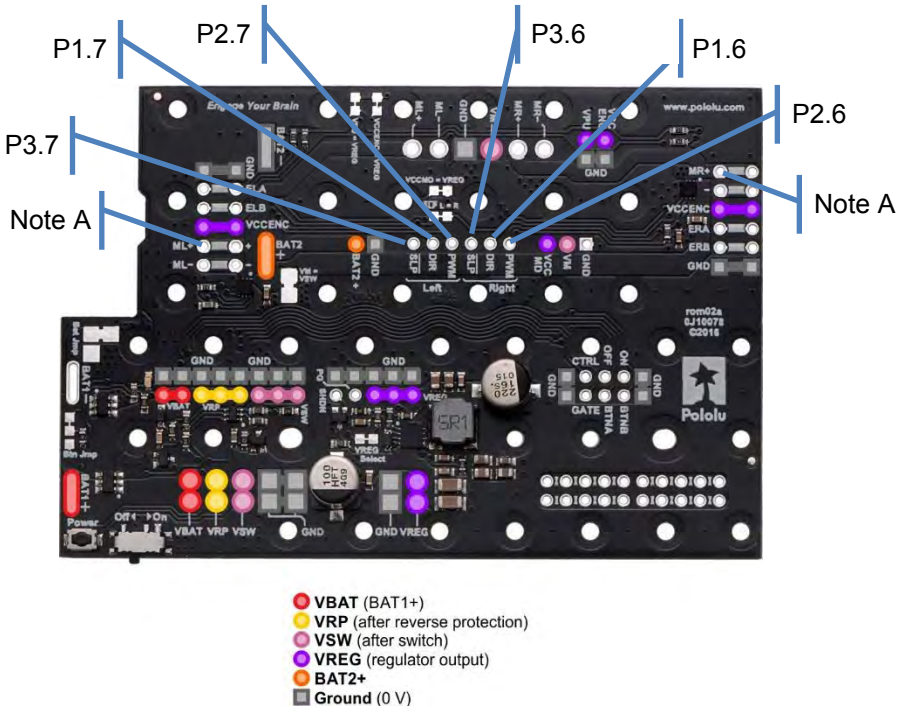


Figure 3. Motor Driver and Power Distribution Board for Romi Chassis. Refer back to Module 5 for power and ground connections. See instructions for Romi chassis for how to connect motors and encoders to the board.

Figure 4 shows a partially completed wheel assembly, and Figure 5 shows one completed wheel assembly.

Warning: Disconnect the VREG↔+5V wire when the LaunchPad USB cable is connected to the PC. Connect the VREG↔+5V wire when the robot is running on battery power. This way the motors always get power from the batteries, and never get power from the USB.

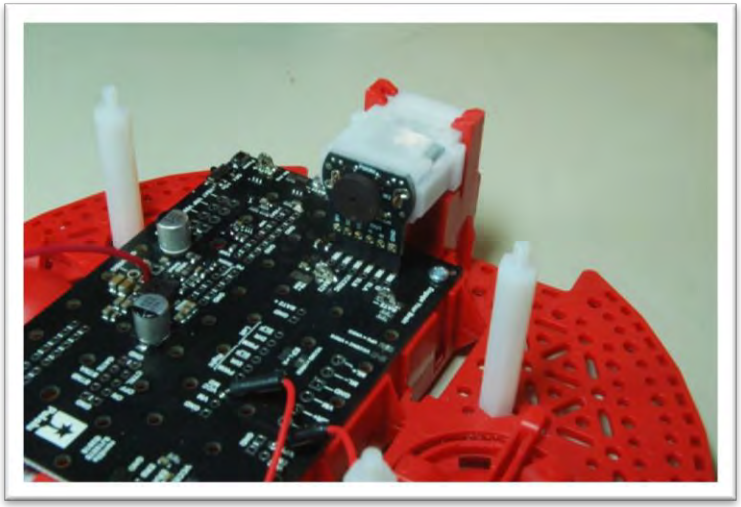


Figure 4. Partially completed wheel assembly.

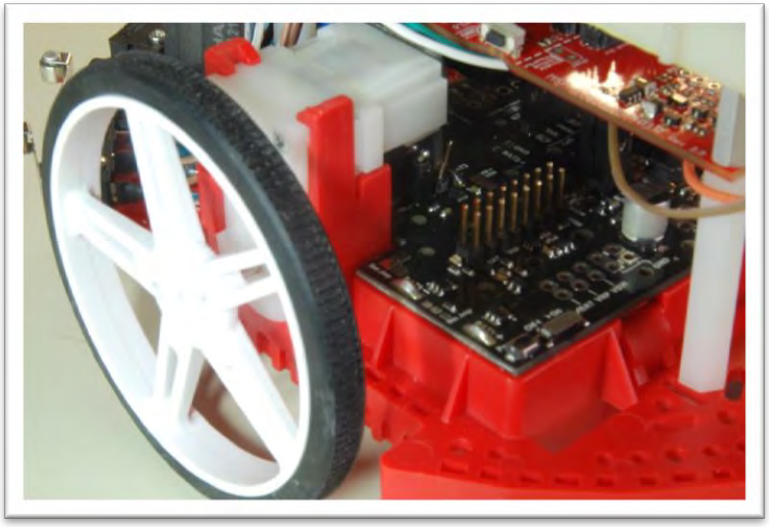


Figure 5. Completed wheel assembly.

12.4 System Development Plan

12.4.1 Low-level software driver

You will start with creating a suite of software functions that control the two wheels on the robot. The frequency of the PWM signal sent to both motors should be 100 Hz (10ms). In this lab, we will keep the duty cycle the same for both motors as well. In the next module, we will use the hardware timer so each motor will have its own duty cycle. To stop the motors you will stop the PWM and set the **nSleep** signal to 0. Use the simple approach of Lab 9 to create the PWM signals. The prototypes for the driver are:

void Motor_InitSimple(void);
Initializes the 6 GPIO lines and puts driver to sleep
Returns right away

void Motor_StopSimple(void);
Stops both motors, puts driver to sleep
Returns right away

void Motor_ForwardSimple(uint16_t duty, uint32_t time)
Drives both motors forward at **duty** (100 to 9900)
Runs for **time** duration (units=10ms), and then stops
Stop the motors and return if any bumper switch is active
Returns after **time***10ms or if a bumper switch is hit

void Motor_BackwardSimple(uint16_t duty, uint32_t time)
Drives both motors backward at **duty** (100 to 9900)
Runs for **time** duration (units=10ms), and then stops
Stop the motors and return if any bumper switch is active
Returns after **time***10ms or if a bumper switch is hit

void Motor_LeftSimple(uint16_t duty, uint32_t time)
Drives just the left motor forward at **duty** (100 to 9900)
Right motor is stopped (sleeping)
Runs for **time** duration (units=10ms), and then stops
Stop the motor and return if any bumper switch is active
Returns after **time***10ms or if a bumper switch is hit

void Motor_RightSimple(uint16_t duty, uint32_t time)
Drives just the right motor forward at **duty** (100 to 9900)
Left motor is stopped (sleeping)
Runs for **time** duration (units=10ms), and then stops
Stop the motor and return if any bumper switch is active
Returns after **time***10ms or if a bumper switch is hit

Lab: DC motors

12.4.2 Control of the motor

In this part of the lab you will implement the functions to test the motors. Place voltmeters on the VM line (+7.2) and on VREG line (+5V) the first time you power up the wheeled robot. Place the robot on blocks, so the wheels do not touch the ground, and test the low-level motor functions, using a program like **Program12_1**. This allows the motors to spin without the robot moving. With the wheels off the ground, there will be minimal friction, the fastest rotation, and the smallest current.

```
// Driver test
void Pause(void){
  while(LaunchPad_Input()==0);  // wait for touch
  while(LaunchPad_Input());     // wait for release
}
int Program12_1(void){
  Clock_Init48MHz();
  LaunchPad_Init(); // built-in switches and LEDs
  Bump_Init();      // bump switches
  Motor_InitSimple();     // your function
  while(1){
    Pause();
    Motor_ForwardSimple(5000,2000);  // your function
    Pause();
    Motor_BackwardSimple(5000,2000); // your function
    Pause();
    Motor_LeftSimple(5000,2000);     // your function
    Pause();
    Motor_RightSimple(5000,2000);    // your function
  }
}
```

Use an oscilloscope to observe the motor signals motor board (MR+, MR-, ML+, ML-) during operation. *You should see voltage versus time. The voltage difference between MR+ and MR- is the applied voltage to the motor.*

Note: As mentioned in Lab 9, using software delays to create PWM consumes all of the processor time. In the next module, we will use the hardware timers on the microcontroller to create the two PWM outputs. In this way, software needs to execute only when it wishes to change the duty cycle or change direction.

12.4.3 Behavior

From an electrical standpoint the motor has three components, resistance (caused by the long wires), inductance (caused by the coiled wires) and electro motive force (emf -voltage caused by the coupling between mechanical and electrical forces). Begin by measuring the resistance of the motor when all power is turned off and the motor is not spinning. Let **R** be this static resistance. Assuming a voltage of 7V, use Ohm's Law to calculate the expected current.

In this section, you will measure actual voltage (**V** in volts), current (**I** in amps), and speed (**s** in rpm) as a function of the **duty** parameter (2000 to 8000). If you place the robot on blocks and attach string/yard/tape to a wheel you can both see and hear the wheel turn. First you will use a stopwatch to count the number of rotations in a fixed time (e.g., 60 seconds).

There are two approaches to measuring motor **current** (**I**). One approach is to remove the batteries and connect a bench supply (which allows you to set the voltage to 7.2V and measure the current) to power the robot. The second approach is to place a current meter in the loop between the batteries and the robot. For example, you can make a 3-layer stack of wire-insulator-wire, and place this stack between the contacts in the battery compartment. You then can place the current meter on the two wires. You can measure motor **voltage** (**V**) with the oscilloscope and verify which duty cycle is active. You will first measure current to the robot with the motors stopped, and then you will measure voltage, current, speed required to spin one motor. The difference in these two current measurements is the current to the motor. You can use a program like **Program12_2** to collect data.

```
// Voltage current and speed as a function of duty cycle
int Program12_2(void){ uint16_t duty;
  Clock_Init48MHz();
  LaunchPad_Init();   // built-in switches and LEDs
  Bump_Init();        // bump switches
  Motor_InitSimple(); // initialization
  while(1){
    for(duty=2000; duty<=8000; duty=duty+2000){
      Motor_StopSimple();   // measure current
      Pause();
      Motor_LeftSimple(duty,6000);  // measure current
    }
  }
}
```

Lab: DC motors

Make a table and graphs of voltage, current, power, emf, and speed as a function of duty cycle. Calculate **emf** as

emf = V – I*R

where **V** is the measured motor voltage, **I** is the measured motor current, and **R** is the static resistance of the motor. Under normal operating conditions, emf will be negative, meaning it draws more current than predicted using the static resistance. Calculate power as

P = V * I * duty/10000

Describe the general behavior of the motor.

Perform a maximum speed test using **Program12_3**. First measure the rotational speed of the motors when the robot is on blocks, and then repeat the measurement when the robot is on the ground.

```
int Program12_3(void){
  Clock_Init48MHz();
  LaunchPad_Init();   // built-in switches and LEDs
  Bump_Init();        // bump switches
  Motor_InitSimple(); // initialization
  while(1){
    Pause();
    Motor_ForwardSimple (9900,1500); // max speed 15 s
  }
}
```

12.5 Troubleshooting

Motors not do spin or gets hot:

- Remove power and double check the connections.
- Review steps in Lab 5.
- Recharge the batteries.
- Verify the six signals from the LaunchPad to the motor board using a voltmeter, an oscilloscope or a logic analyzer.

One motor spins faster than the other:

- It is normal for the motor speeds to be ±20% of each other
- Check for friction on the slower motor

12.6 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. The goal of this module is for you to experience voltage, current, and power as they relate to DC motors.

- How does friction affect motor current?
- In this lab, we do not set the speed or the current. Rather, we set just the voltage and duty cycle. Why is it difficult in this lab for the robot to go straight?
- How does the two H-bridges allow the robot to turn, to back up?
- How does the software adjust power delivered to the motors?
- In what two ways could software cause the robot to turn?

12.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. For example,

- If you do not have the Pololu motor board, you could build your own H-bridge circuits to control the motors on the robot. In particular, you could build two H-bridges described in lecture using the L293. If you build your own H-bridge please test it before attaching the motors and before attaching the microcontroller.
- An impossible challenge would be to try to write software that makes the robot travel in a square pattern. Basically, repeat this two-step process: 1) go straight for a fixed amount of time; 2) turn left 90 degrees. It will not be possible. However, it will be instructive to determine why the effort fails.

12.8 Which modules are next?

There are two major limitations to the robot conceived in this lab. 1) the software consumes all the processor time, and 2) the speed of the motors depends on many factors most of which cannot be predicted in advance. Over the remaining labs we will solve these limitations.

Module 13) Use timers to create PWM signals, and use interrupts to manage multiple software tasks
Module 15) Use the ADC to interface distance sensors. Two distance sensors can be used to drive the robot at a fixed distance and fixed angle to the wall.
Module 16) Interface tachometers (Romi Encoder Pair Kit) and use timer capture to measure the speeds of each wheel directly.
Module 17) Combine modules 12, 13, and 16 to create a control system that does spin the motors at a desired speed.

12.9 Things you should have learned

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

- Understand voltage, current, and power to a motor.
- Be able to use PWM output to adjust power to the motors.
- Understand basic operation and purpose of an H-bridge.
- Know how to write and test a low-level software driver.

Module 13

Introduction: Timers

Introduction: Timers

Educational Objectives:

UNDERSTAND Timers and their uses in embedded systems
INTERFACE The DC motors using hardware PWM
CREATE Multi-threaded software using multiple periodic interrupts
DESIGN Robot commands that move forward, turn left, turn right, and move backward

Prerequisites (Modules 9, 10, 12)
- Pulse width modulation (Module 9)
- Periodic interrupts using SysTick (Module 10)
- Mechanical and electrical interfaces of motors (Module 12)

Recommended reading materials for students:
- Volume 1 Sections 8.7, and 9.7
  Embedded Systems: Introduction to the MSP432 Microcontroller, ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or

- Volume 2 Sections 6.2, 6.3, and 6.5
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

This module, together with the last (Module 12), will develop the robot so it moves. Back in Module 9 you created software using pulse width modulation that dimmed an LED. You will now replace software-generated PWM with hardware-generated PWM. More specifically, you will configure the timer hardware on the MSP432 microcontroller. This will allow the system to adjust the power delivered to the DC motors on the robot with very little software overhead. Software will initialize the times, setting the PWM period and initial duty cycle. The hardware timers will automatically create the PWM outputs. Software needs to execute only when the system wishes to change the applied power or change the direction.

Back in Module 10, you created two threads: main program and SysTick ISR. In this module, you will use the hardware timers to create an additional periodic thread. Having multiple threads allows you to increase the complexity of the system in a modular way.

MSP432 microcontrollers have timers that are separate and distinct from SysTick. Input capture mode is used to make time measurements on input signals (Module 16), measuring the period from the tachometer (Lab 16). The MSP432 microcontroller has four **General Purpose Timer Modules** called Timer_A. Each timer has one 16-bit timer and seven associated capture/compare registers.

In this lab, you will use Timer A0 to create two PWM outputs for the motor interface, and you will use the Timer A1 to create an additional periodic interrupt that can be used by robot explorer. You will use Timers A2 and A3 later in Module 16 to interface the two tachometers.

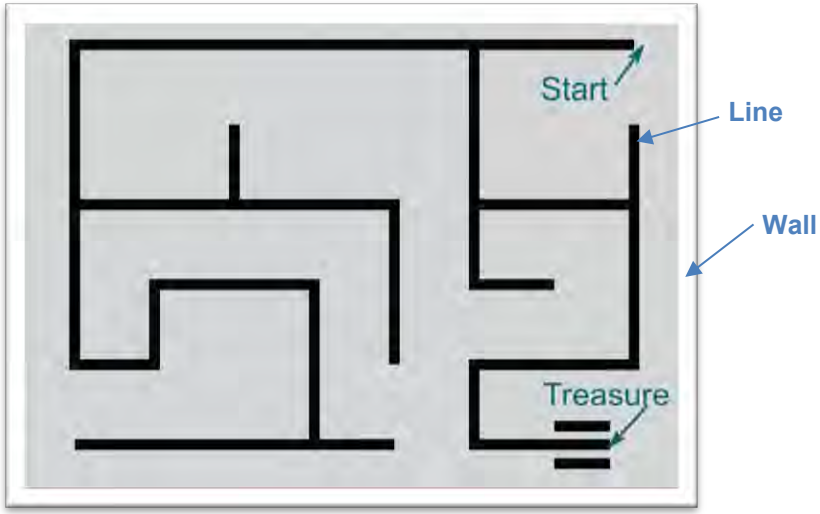


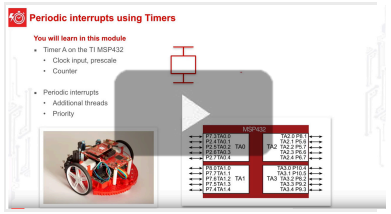
Figure 1. (From Lab 6) After this lab you could create a robot explorer that finds its way out of a maze, using just the line sensors and bump sensors. Similarly, you could create a robot explorer that follows a line.

13. TI-RSLK Module 13 – Timers

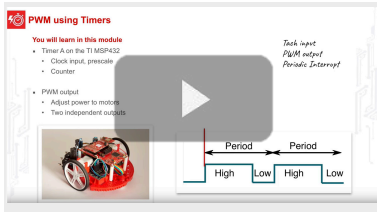
In this module, you will write software that uses the timers to create PWM outputs. Using timers for PWM and period interrupts provide mechanisms to grow the complexity of the robot system.

Optionally, download all the curriculum documents for Module 13.

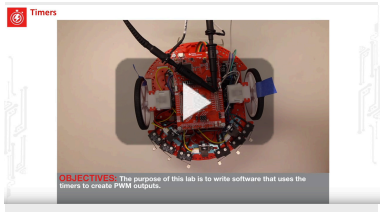
13.1 TI-RSLK Module 13 - Lecture video part I - Timers - Periodic interrupt
In this module you will learn how to interface the DC motors using hardware PWM.



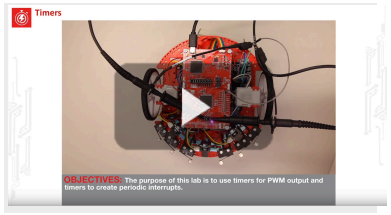
13.2 TI-RSLK Module 13 - Lecture video part II - Timers - Pulse width modulation
In this module you will learn how to interface the DC motors using hardware PWM.



13.3 TI-RSLK Module 13 - Lab video 13.1 – Timer generated PWM outputs to spin motors
Understand timers and their uses in embedded systems and interface the DC motors using hardware PWM.



13.4 TI-RSLK Module 13 - Lab video 13.2 – Interrupt latency
Understand timers and their uses in embedded systems.



Module 13

Lab 13: Timers

Lab: Timers

13.0 Objectives

The purpose of this lab is to develop the software needed to spin the motors. The software can vary the **electrical power** delivered to each motor using **pulse width modulation** (PWM). In this module,

1. You will learn the MSP432 Timer_A module.
2. You will configure Timer A0 to create two PWM outputs.
3. You will configure Timer A1 to create an additional periodic interrupt.
4. You will develop low-level robot commands for movement.

Good to Know: PWM is an effective and efficient means for the microcontroller to affect its world. It is effective because setting the timer reload value to 15000 will create an output with essentially 14 bits of precision. It is efficient because the cost of the timer and digital switching circuit (DRV8838) is much less than an equivalent analog amplifier.

13.1 Getting Started

13.1.1 Software Starter Projects

Look at these three projects:
PWMSine (uses a PWM and a timer to create a sine wave output)
PeriodicTimerA0Ints (uses Timer_A0 to create a periodic interrupt)
Lab13_Motors (starter project for this lab)

Note: You will not be able to run the PeriodicTimerA0Ints project on the robot because this project uses Timer_A0, and you need to use Timer_A0 for the robot's two PWM outputs.

13.1.2 Student Resources (in datasheets directory)

MSP432P4xx Technical Reference Manual, Timer_A (SLAU356)
MSP432P401R Datasheet, msp432p401m.pdf (SLAS826)
MotorDriverPowerDistribution.pdf Data sheet for power board
Pololu Romi Chassis User's Guide.pdf How to build the robot
drv8838.pdf data sheet for the H-bridge driver

13.1.3 Reading Materials

Volume 1 Sections 8.7, and 9.7
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 6.2, 6.3, and 6.5
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

13.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	Romi Chassis Kit - Red	Pololu	3502
1	Motor Driver and Power Distribution Board for Romi	Pololu	3543
1	Romi Encoder Pair Kit, 12 CPR* optional	Pololu	3542
2	Rechargeable Battery, Pack of 4, Metal Hydride 1300 mAh, 1.2V, AA	Energizer	626831
4	1.375in 4-40 Nylon standoff	Keystone	4809
2	0.187in 4-40 metal nut	Keystone	4694
6	0.5in 4-40 Nylon machine screw	Pololu	1962

Lab: Timers

13.1.5 Lab equipment needed
Oscilloscope (one or two channels at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling

Warning: Disconnect the VREG↔+5V wire (the one between the MBDB and the LaunchPad) when the LaunchPad USB cable is connected to the PC. Connect the VREG↔+5V wire when the robot is running on battery power. This way the motors always get power from the batteries, and never get power from the USB.

13.2 System Design Requirements

The first goal of this lab is to write software that can adjust the applied power to the two motors. You will create PWM outputs on the P2.6 and P2.7 pins, which are connected to the PWML and PWMR of the MDPDB (EN input to the DRV8838). The period of both outputs should be fixed at 10 ms (100 Hz). However, the software should be able to independently set the duty cycle of the EN pin to each motor from 0 to 14,998 (0 to 99.99%). At 100 Hz, the motor will not respond to individual highs and lows; rather, the motors will respond to the average level. More specifically, the delivered power will be

$$\mathbf{P} = \mathbf{V} * \mathbf{I} * \mathbf{duty}/15000;\ 0 \le \mathbf{duty} \le 14998$$

where **V** is the voltage and **I** the current, as measured previously in Lab 12.

The second goal of this lab is create an additional periodic interrupt using Timer_A1. The high-level main program will initialize this periodic interrupt using a function pointer at run time, providing for abstraction and code reuse.

Similar to Lab 12, the outcome of this lab is a system that drives in a straight line until one of the bump sensors detects a collision. However, contrary to Lab 12, this solution will require very little software overhead.

13.3 Experiment set-up

Same as Lab 12. Refer to the data sheet of the DRV8838 to see how the software output values to these six signals affect motor behavior, Figure 1.

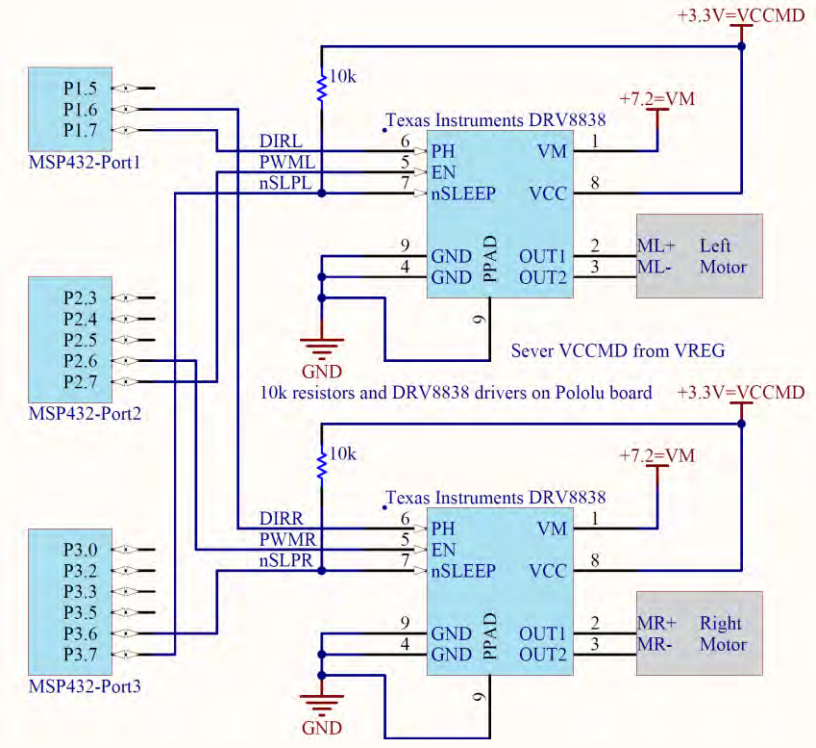


Figure 1. Interface circuit.

LaunchPad	MDPDB	DRV8838	Description
P1.6	DIRR	PH	Right Motor Direction
P3.6	nSLPR	nSLEEP	Right Motor Sleep
P2.6	PWMR	EN	Right Motor PWM
P1.7	DIRL	PH	Left Motor Direction
P3.7	nSLPL	nSLEEP	Left Motor Sleep
P2.7	PWML	EN	Left Motor PWM

13.4 System Development Plan

13.4.1 Low-level software driver

Replace the suite of software functions built in Lab 12 with functions that use Timer_A0 to create the two PWM outputs. This suite of functions will control the two wheels on the robot. When active the PWM to both motors will be 100 Hz (10 ms), but have independent duty cycles. The prototypes for the driver are:

void Motor_Init(void);
Initializes the 6 lines and Timer A0 and puts driver to sleep
Returns right away

void Motor_Stop(void);
Stops both motors, puts driver to sleep
Returns right away

void Motor_Forward(uint16_t leftDuty, uint16_t rightDuty)
Drives left motor forward at **leftDuty** (0 to 14,998)
Drives right motor forward at **rightDuty** (0 to 14,998)
The motors run until software issues another command
Returns right away

void Motor_Backward(uint16_t leftDuty, uint16_t rightDuty)
Drives left motor backward at **leftDuty** (0 to 14,998)
Drives right motor backward at **rightDuty** (0 to 14,998)
The motors run until software issues another command
Returns right away

void Motor_Left(uint16_t leftDuty, uint16_t rightDuty)
Drives left motor backward at **leftDuty** (0 to 14,998)
Drives right motor forward at **rightDuty** (0 to 14,998)
The motors run until software issues another command
Returns right away

void Motor_Right(uint16_t leftDuty, uint16_t rightDuty)
Drives left motor forward at **leftDuty** (0 to 14,998)
Drives right motor backward at **rightDuty** (0 to 14,998)
The motors run until software issues another command
Returns right away

13.4.2 Motor Testing

Place voltmeters on the VM line (+7.2) and on VREG line (+5V) of the MDPD board while debugging the motor software. Place the robot on blocks, so the wheels do not touch the ground, and test the low-level motor functions, using a program like **Program13_1**.

```
// Driver test
void TimedPause(uint32_t time){
  Clock_Delay1ms(time);  // run for a while and stop
  Motor_Stop();
  while(LaunchPad_Input()==0);  // wait for touch
  while(LaunchPad_Input());     // wait for release
}
int Program13_1(void){
  Clock_Init48MHz();
  LaunchPad_Init(); // built-in switches and LEDs
  Bump_Init();      // bump switches
  Motor_Init();     // your function
  while(1){
    TimedPause(4000);
    Motor_Forward(7500,7500);  // your function
    TimedPause(2000);
    Motor_Backward(7500,7500); // your function
    TimedPause(3000);
    Motor_Left(5000,5000);     // your function
    TimedPause(3000);
    Motor_Right(5000,5000);    // your function
  }
}
```

Place the robot on the ground and try to adjust each of the 7500 parameters in the calls to ***Motor_Forward*** and ***Motor_Backward*** so robot moves in a straight line. Adjust the 5000 parameters in the calls to ***Motor_Left*** and ***Motor_Right*** and the 3000 parameters to **Pause**, so robot turns 90 degrees.

Note: Adjusting these parameters to run the robot open loop will be virtually impossible. Asking you to try to solve an impossible problem will motivate the need for inputs and create a closed loop controller.

13.4.3 Periodic Interrupt

Write the software to create an additional periodic interrupt using Timer_A1. If you use the 12 MHz SMCLK and divide by 24, the 16-bit timer will clock at 500 kHz. At this clock rate, the slowest interrupt that can be created is about 130 ms (65535*2μs). You can use a program like **Program13_2** to test this driver. Notice the use of bit-banding to remove the critical section that would normally occur with a read-modify-write sequence on a shared global.

```
// Test of Periodic interrupt
#define REDLED (*((volatile uint8_t *)(0x42098060)))
#define BLUELED (*((volatile uint8_t *)(0x42098068)))
uint32_t Time;
void Task(void){
  REDLED ^= 0x01;       // toggle P2.0
  REDLED ^= 0x01;       // toggle P2.0
  Time = Time + 1;
  REDLED ^= 0x01;       // toggle P2.0
}
int Program13_2(void){
  Clock_Init48MHz();
  LaunchPad_Init();  // built-in switches and LEDs
  TimerA1_Init(&Task,50000);  // 10 Hz
  EnableInterrupts();
  while(1){
    BLUELED ^= 0x01; // toggle P2.1
  }
}
```

Use a dual trace scope to observe both P2.0 (interrupt thread) and P2.1 (main thread). Trigger on the interrupt signal and use the gap in the oscillations on P2.1 to estimate the time required to service the Timer A1 interrupt.

After testing the PWM and Timer A1 separately, combine them into one software system that runs the robot like Program 13.1, but uses the periodic interrupt to check the bump switches, stopping the robot on a collision.

13.5 Troubleshooting

PWM or interrupts are the incorrect period:

- Check the source of the timer clock.
- Make sure the processor is running at 48 MHz.

PWM output does not occur:

- Run the **PWMSine** project to see if the hardware is ok.
- Use the debugger to make sure the Timer_A0 registers are set.

Interrupts do not occur:

- Run the **PeriodicTimerA0Ints** project and use the debugger to observe the Timer A0 registers. Run your program and observe the Timer A1 registers
- Use the debugger to observe the registers in the NVIC
- Make sure the I-bit is clear, by calling **EnableInterrupts()**;

Lab: Timers

13.6 Things to think about

In this section, we list questions to consider after completing this lab. These questions are meant to test your understanding of the concepts in this lab. The goal of this module is for you to understand Timer_A and its use for PWM and periodic interrupts.

- How does the software select the input clock for Timer_A?
- What does the prescaler do for Timer_A? Why is the prescaler important (i.e., what happens when you change the prescale?)
- What is the precision of the PWM generated in this lab?
- What would happen if the main program in **Program13_2** while loop executed **P2->OUT ^= 0x04;** instead?
- How could you use Timer A1 to perform periodic tasks once a second?
- What is a function pointer? Why are function pointers used in this lab?

13.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. For example,

- If you do not have the Pololu motor board, you will have to change the way your software operates. Luckily, it is possible to create PWM outputs on any of the P2.4, P2.5, P2.6 or P2.7.
- It is now possible to combine Lab 7 (FSM), Lab 12 (motors) and this lab to create a robot that follows a line.

13.8 Which modules are next?

The major limitation to the robot conceived in this lab is the speed of the motors depends on many factors most of which cannot be predicted in advance. Therefore the system must deploy sensors to determine its state. Over the remaining labs we will solve these limitations.

Module 15) Use the ADC to interface distance sensors. Two distance sensors can be used to drive the robot at a fixed distance and fixed angle to the wall.
Module 16) Interface tachometers (Romi Encoder Pair Kit) and use timer capture to measure the speeds of each wheel directly.
Module 17) Combine modules 12, 13, and 16 to create a control system that spins the motors at a desired speed.

13.9 Things you should have learned

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

- Understand voltage, current, and power to a motor.
- Be able to use PWM output to adjust power to the motors.
- Understand basic operation and purpose of an H-bridge.
- Know how to write and test a low-level software driver.

Module 14

Introduction: Real-time Systems

Introduction: Real-time Systems

Educational Objectives:

UNDERSTAND how to use priority interrupts for creating real-time systems
EXPLORE different techniques to interface switches
LEARN how to generate port interrupts on the GPIO input pins
DESIGN, BUILD & TEST A SYSTEM
Create a real-time system for collision detection

Prerequisites (Modules 8, 9, 10, and 13)
- Switch interfacing (Module 8)
- Time delays (Module 9)
- SysTick periodic interrupts (Module 10)
- Timer_A periodic interrupts (Module 13)

Recommended reading materials for students:
- Volume 1 Sections 9.1, 9.2, 9.3, 9.4, and 9.5
  Embedded Systems: Introduction to the MSP432 Microcontroller ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or
- Volume 2 Sections 4.5, 5.1, 5.2, 5.3, 5.4, and 5.5
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

Previously we defined a **real-time system** as one with bounded latency. In other words, the **latency**, which is the time between when a service is requested and the time when service is initiated, is always less than small and acceptable limit. Depending on the situation, we could alternately define real time as having a bounded **response time.** For example, for collision detection on the robot, we define response time as the time between a collision (bump sensor hardware edge signifying a service is requested) and the time when the motors are stopped (service is complete). To make it real time, we will configure the bump sensors to request an interrupt on touch.

The basic approach to a system requiring multiple software tasks is to deploy multithreading. One software **thread** is the traditional main program, which runs most of the time. This thread will implement high-level strategy. Interrupts will be used to create additional threads. The SysTick periodic interrupt will measure data from the line sensor. In Module 13, we studied how to execute periodic tasks using Timer_A. In this module, we will learn how use edge-triggered interrupts generated by I/O pins.

Any of the pins on Ports 1 – 6 can request an interrupt. We can configure the interrupt request on a rise or a fall of the input signal. If the bump switches are interfaced with negative logic, then a falling edge signifies a collision has occurred. Interrupts communicate with other threads via global variables. When deploying multiple interrupts we use priority to sort out the order of service if multiple events coincide. This collision detection is a very high priority task and hence we will configure it as a high priority event.

In this lab, the collision will cause the motors to stop and also set a global error flag. The main program will recognize this event, and then do something appropriate, like back up the robot turn 90 degrees and continue forward again. In the lab, there will be an option to solve a very simple systems-level robotic challenge.

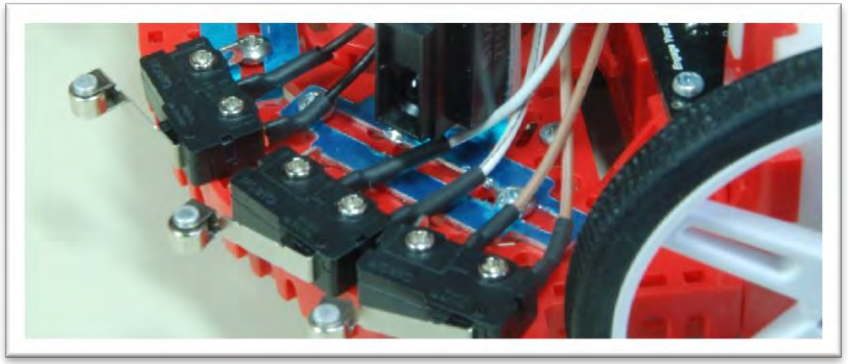


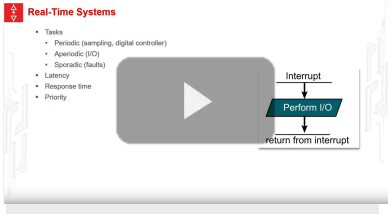
Figure 1. Bump sensors, positioned at the front of the robot.

14. TI-RSLK Module 14 – Real-time systems

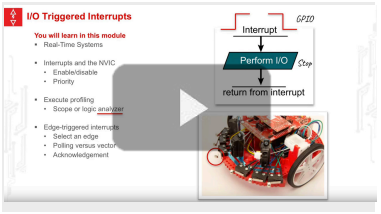
This module demonstrates how to use priority interrupts for creating real-time systems. As your robot system becomes more complex, period interrupts are one way to combine multiple threads onto one microcontroller.

Optionally, download all the curriculum documents for Module 14.

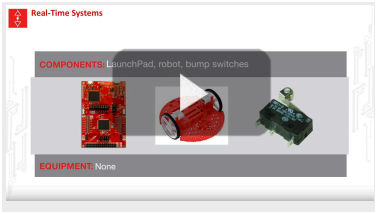
14.1 TI-RSLK Module 13 - Lecture video part I - Timers - Periodic interrupt
In this module you will learn how to interface the DC motors using hardware PWM.



14.2 TI-RSLK Module 13 - Lecture video part II - Timers - Pulse width modulation
In this module you will learn how to interface the DC motors using hardware PWM.



14.3 TI-RSLK Module 13 - Lab video 13.1 – Timer generated PWM outputs to spin motors
Understand timers and their uses in embedded systems and interface the DC motors using hardware PWM.



Module 14

Lab 14: Real-time Systems

Lab: Real-time Systems

14.0 Objectives

The purpose of this lab is to interface bump sensors to detect collision, which will be one task that the robot will need to explore its world; see Figure 1.

1. You will use edge-triggered interrupts to detect collisions.
2. You will use shared global variables to communicate between threads.
3. You will use priority to define order of execution when servicing multiple concurrent events.
4. You will profile the time to service an interrupt.
5. You will combine this lab with previous labs to solve a problem.

Good to Know: Interrupts are extremely important for embedded systems, providing a mechanism to implement real-time behavior and multi-threading.

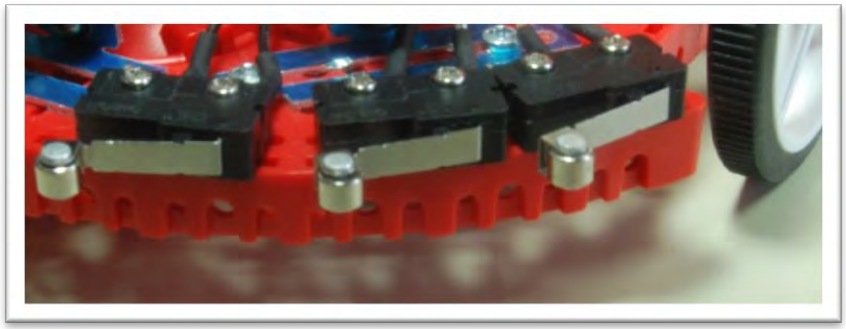


Figure 1. Bump sensors positioned at the front of the robot.

14.1 Getting Started

14.1.1 Software Starter Projects

Look at these three projects:
EdgeInterrupt (edge-triggered interrupt on P1.1 and P1.4),
Lab13_Timers (your solution to lab 13)
Lab14_EdgeInterrupts (starter project for this lab)

14.1.2 Student Resources (in datasheets directory)

Meet the MSP432 LaunchPad (SLAU596)
MSP432 LaunchPad User's Guide (SLAU597)
Polulu_BumpSwitch_1404.png, mechanical drawing of switch
QTR-8x.pdf, line sensor datasheet

14.1.3 Reading Materials

Volume 1 Sections 9.1, 9.2, 9.3, 9.4, and 9.5
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 4.5, 5.1, 5.2, 5.3, 5.4, and 5.5
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

Good to Know: Edge-triggered interrupts is a useful feature of microcontrollers that are used commonly in embedded systems. In general, external events can be signified as a change in status. Examples include danger, power failure, temperature overload, system faults. If the status is an external digital logic signal, it can be connected to a GPIO input, and the system can request an interrupt on a rising or falling edge of that signal.

14.1.4 Components needed for this lab (combination of Labs 10 and 12)

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
6	Bump switches	Pololu	#1404
1	QTR-8RC Reflectance Sensor Array	Pololu	#961
12	0.5in 2-56 screw	Pololu	#2715
12	2-56 nut	Multicomp	SPC21805
1	Romi Chassis Kit - Red	Pololu	3502

1	Motor Driver and Power Distribution Board for Romi	Pololu	3543
1	Romi Encoder Pair Kit, 12 CPR* optional	Pololu	3542
2	Four Eneloop AA 2100 Cycle Ni-MH	Panasonic	BK-3MCCA4BA
4	0.25in 4-40 machine screw	Digikey	36-9900-ND
4	1.375in 4-40 Nylon standoff	Digikey	36-4809-ND

Table 1 Parts needed for this lab

14.1.5 Lab equipment needed

Oscilloscope (one or two channels at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling

14.2 System Design Requirements

The first goal of this lab is to use edge-triggered interrupts to detect collisions by the bump sensors while the robot is moving. A collision event should cause an interrupt, and reading the status of the bumper switches should occur in the **interrupt service routine** (ISR).

In the previous labs, we suggested you make the periodic interrupt used for the line sensor (see SysTick Interrupt in Module 10) a high priority because it is a real time measurement. However, once we integrate labs together, the collision task should have a priority even higher priority than the periodic measurements because a collision represents a danger condition that requires immediate service. In lab 10, you attached the bump sensors to the robot and interfaced them to the microcontroller. In this lab, you will shift the software servicing of the sensors from a periodic interrupt to an event-driven trigger.

The second goal of this lab is integrate components into a single hardware software solution that performs a simple but integrated task. This integrated task must include the edge-triggered interrupt interface of the bump sensors.

Feel free to adapt/combine solutions/sensors from previous labs. For example, you could explore a fenced in arena, as shown in Figure 2. When the robot collides into the wall, it could back up a little, turn, and then continue forward.

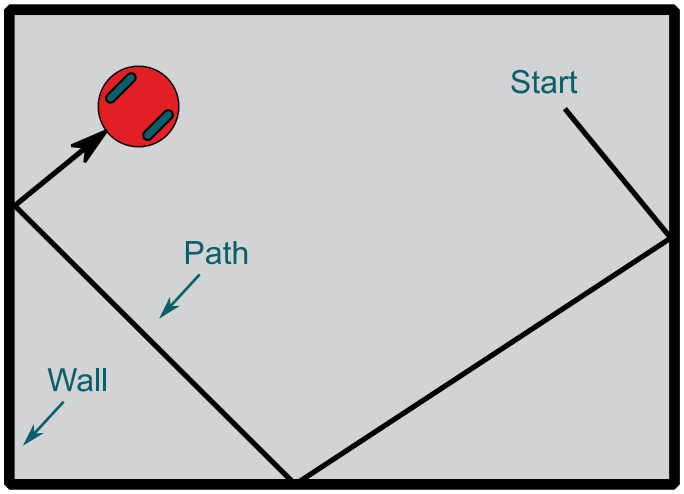


Figure 2. A possible integrated maze is to explore inside a box.

This high-level strategy should be performed separate from the edge-triggered ISR caused by the bump switch touch edge interrupt. For example, you could define three threads

- Periodic Timer_ A1 interrupts to run the high-level strategy
- Edge triggered interrupts for collisions
- Main program initializes and then does nothing in the loop.

14.3 Experiment set-up

You interfaced the QTR-8RC line sensor back in Lab 6. You interfaced the bump sensors back in Lab 10. Figure 3 shows one possible placement for six sensors. Figure 4 shows a simple electrical circuit for interfacing the switches.

Warning: *TI MSP432 pins are not 5V tolerant; you must power the line sensor and bump sensors with +3.3V.*

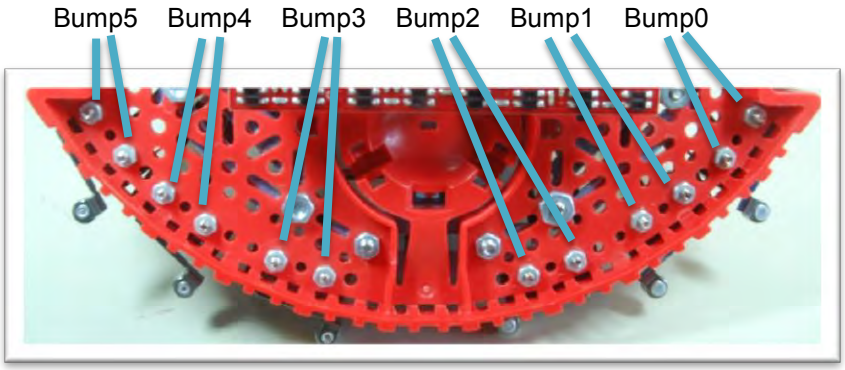


Figure 3. Bump sensors attached to the front of the robot (bottom view).

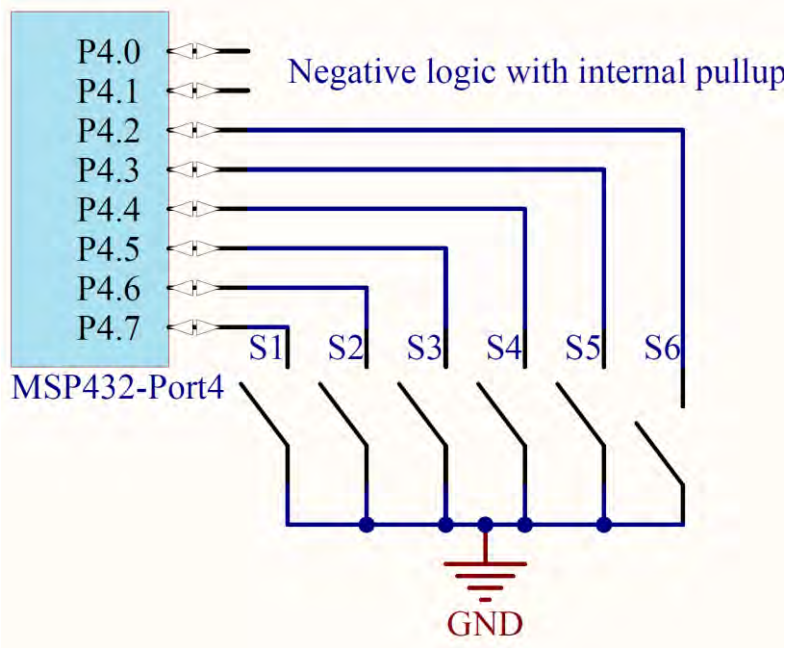


Figure 4. One possible interface circuit for the bump sensors. Using Port 4, will conflict with the CC3100/CC3120 wifi module. However, only Ports 1 – 6 can trigger interrupts.

14.4 System Development Plan

14.4.1 Interface the switches and motors

If you didn't interface the bump sensors as part of Lab 10, follow the Lab 10 directions and attach the sensors to the robot and interface the signals to the microcontroller. If you didn't interface the motors as part of Lab 12, follow the Lab 12 directions and attach the motors to the robot and interface the signals to the microcontroller.

14.4.2 Develop and debug the edge-triggered interrupts

You will write one function to initialize the bump sensors, **BumpInt_Init()**. This function configures the appropriate port pins, enables internal resistors as needed, and **enables edge-triggered interrupts**. You need a way to integrate the low-level device driver code with the high-level robotic system. One way is to place the ISR at the high level. This is a simple approach, but it does intertwine high-level with low-level code. A more elegant solution is to use a hook or function pointer. The user-supplied function is passed from high level to low level dynamically, when the interface is initialized. This high-level function will be called on a collision from your ISR that handles the edge-triggered event. To provide additional functionality, your ISR will pass a 6-bit value from the sensors.

```
void BumpInt_Init(void(*task)(uint8_t));
```

Note: In previous labs, you handled collisions within a periodic ISR. If the interrupt period is 10ms, the average latency is 5ms and the worst case latency of a collision event will therefore be 10ms. When running as a high priority edge-triggered interrupt, the latency will be on the order of 1 µs.

You designed and tested the function Motor_Stop as part of Lab 13. For more information on the motors, refer back to Labs 12 and 13. For example, if the robot needs to stop, you define a function

```
uint8_t CollisionData, CollisionFlag;  // mailbox
void HandleCollision(uint8_t bumpSensor) {
  Motor_Stop();
  CollisionData = bumpSensor;
  CollisionFlag = 1;
}
```

HandleCollision is defined within the high level software. When you initialize the bump sensors you pass in a pointer to this high-level routine.

```
BumpInt_Init(&HandleCollision);
```

This function will be called from an ISR, so its execution time should be short and bounded. In other words, please avoid long delay loops in the ISRs.

14.4.3 Profiling

Use an oscilloscope and an unused pin to measure the latency of the collision detector. One channel of the scope shows the falling edge (collision) and a second channel shows when the ISR is run. You can use the triple toggle technique to measure both latency (delay from collision to the start of service) and response time (delay from collision until the motors are stopped).

You will need a real scope or logic analyzer (and not TExaS), because the times will be on the order of microseconds *(TExaS has a time resolution of 100µs)*.

14.4.4 Integrated Robotic System

While debugging the integrated system use the dump techniques learned in Lab 10 to record strategic information during the run. Operate the robot for about a minute, and then observe the debug information to verify robot sensors and actuators operated as intended.

If you run the high-level strategy in a periodic interrupt it will be easy to implement a robot command language like

1. **Back Up** slowly for 1 second
2. **Turn Right** slowly for 5 seconds (90 degrees)
3. **Go Forward** quickly for 1 minute (infinite time)
4. **Repeat steps**

Since this task runs in a periodic interrupt, the software has no loops. More specifically, it has no do-while-loops, no while-loos, and no for-loops. This software structure will be very efficient of processor execution time.

On a collision, you stop and restart this simple set of commands

14.5 Troubleshooting

Bump sensors don't work:

- Check the wiring as described in Figure 3. Figure 3 shows negative logic and internal pull-up resistors. Because there are no external resistors, you do need to configure the internal resistors in software.
- Look at signals with a voltmeter, scope or logic analyzer. You should see the voltage on the microcontroller pin be 0 when pressed and 3.3V when released. The operation of one switch should not affect the signals on the other switches.
- Look at the port registers in the debugger. With the debugger doing periodic updates, and the software running, you should see the port input register change with the switch.

Robot does not operate properly:

- Don't try to solve the entire project all at once. Break the problem into many small components and test each component separately. After two components are tested, combine those two components and test the two components together. Incrementally add components until the system is complete.
- Use profiling techniques to observe CPU utilization. In particular, measure the percentage time each thread requires. Observe if the execution of one task is preventing another thread from running.
- Use the debugging techniques from Lab 10 to be able to observe inputs and outputs in real time while the robot is operating.

14.6 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 interrupt priority affect the behavior of the robot?
- Assume you knew turning about 90 degrees required you to run the motors for 2 seconds. How would you use interrupts to perform this operation?
- What factors affect latency on this robot?
- How would you use interrupts to run the finite state machine from Lab 6, as shown in Figure 7?
- What should you do in the main program to save power?

14.7 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,

- Integrate Lab 11 so debugging information is displayed on the LCD.
- Integrate Lab 10 so debugging information is recorded into Flash ROM.
- Use debugging features within CCS to perform execution profiling.
- Use debugging features within CCS to perform power profiling on the MSP432. However, most of the power used in the robot is delivered from batteries to the motors, so CCS will not be able to monitor this power. To measure total power, you would need to current measurements from Lab 12.

14.8 Which modules are next?

This was our first of many uses of interrupts in this course. The following modules will build on this module:

Module 15) Interface IR distance sensors to the robot using the ADC.
Module 16) Interface tachometers to the microcontroller and use input capture to measure wheel velocity.
Module 17) Combine modules 12, 13 and 16 to develop closed loop motor controllers. In this module you will be able to spin the motors at a constant speed.

14.9 Things you should have learned

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

- Use edge-triggered interrupts to implement multithreading
- Use global variables to communicate between threads
- Perform execution profiling using port pins and a scope
- Perform high-level tasks on the robot

TI-RSLK

Texas Instruments Robotics System Learning Kit
The Maze Edition - Advanced

Texas Instruments

Module 15

Introduction: Data Acquisition Systems

Introduction: Data Acquisition Systems

Educational Objectives:

REVIEW periodic interrupts and the Nested Vector Interrupt Controller
UNDERSTAND how to use the ADC to implement real-time data acquisition systems, observing noise, choosing a sampling rate and thinking about aliasing (undesired frequency components)
EXPLORE the world of digital processing by implementing some simple digital filters
LEARN the Nyquist Theorem and the Central Limit Theorem.
DESIGN, BUILD & TEST A SYSTEM
Create a real-time data acquisition system that measures distance from three IR sensors

Prerequisites (Modules 10, and 13)

- SysTick periodic interrupts and arrays (Module 10)
- Timer_A periodic interrupts (Module 13)

Recommended reading materials for students:

- Volume 1 Sections 10.1, 10.4, and 10.5
  Embedded Systems: Introduction to the MSP432 Microcontroller ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or

- Volume 2 Section 8.4, and Chapter 10
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

An **analog signal** is one that is continuous in both amplitude and time. An **analog signal** is one that is continuous in both amplitude and time. Neglecting quantum physics, most signals in the world exist as continuous functions of time in an analog fashion (e.g., voltage, current, position, angle, speed, force, pressure, temperature, and flow etc.) In other words, the signal has an amplitude that can vary over time, but the value does not instantaneously change. To represent a signal in the digital domain we must approximate it in two ways: **amplitude quantizing** and **time quantizing**. From an amplitude perspective, we will first place limits on the signal restricting it to exist between a minimum and maximum value (e.g., 0 to +3.3V), and second, we will divide this amplitude range into a finite set of discrete values. The **range** of the system is the maximum minus the minimum value. The range has units, such as volts or cm. The **precision** of the system defines the number of values from which the amplitude of the digital signal is selected. Usually precision is given in binary bits. For example, an 8-bit system can uniquely identify 256 different values. The **resolution** is the smallest change in value that is significant. The resolution is given in the same units as the range.

$$\text{range} = \text{resolution} * 2^n, \text{ where n is the precision in bits}$$

The second approximation occurs in the time domain. Time quantizing is caused by the finite sampling interval. In practice we will use a periodic timer to trigger an **analog to digital converter** (ADC) to digitize information, converting from the analog to the digital domain. The **Nyquist Theorem** states that if the signal is sampled with a frequency of f_s, then the digital samples only contain frequency components from 0 to ½ f_s. Conversely, if the analog signal does contain frequency components larger than ½ f_s, then there will be an **aliasing** error during the sampling process. Aliasing is when the digital signal appears to have a different frequency than the original analog signal.

In this lab, we will attach three IR distance sensors to the robot and interface the transducers to the microcontroller using ADC inputs. You will use periodic interrupts to sample the distance to the wall from three positions on the robot. Using the classification algorithm developed in Lab 4, there will be an option to solve a systems-level robotic challenge.

Figure 1. IR distance sensors, positioned at the front of

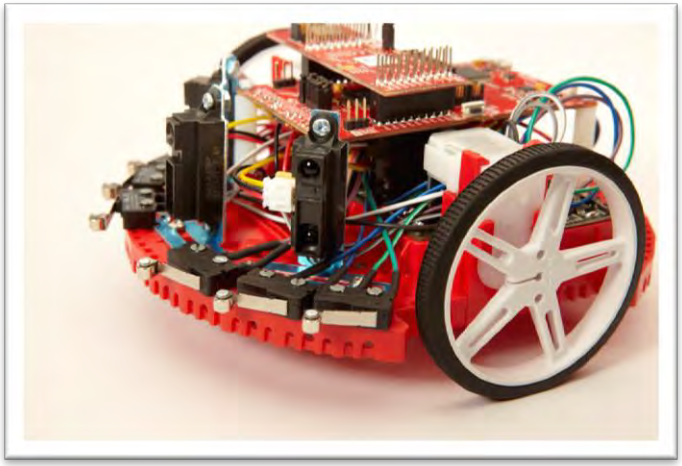


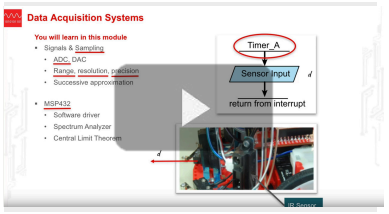
Figure 1. IR distance sensors, positioned at the front of the robot.

15. TI-RSLK Module 15 – Data acquisition systems

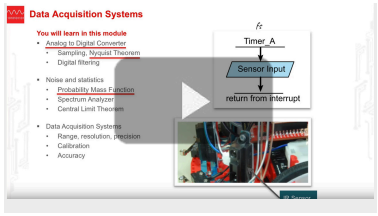
This module will teach you how to interface the infrared distance sensors using the analog-to-digital converter. IR distance sensors are an essential component for solving robot challenges where avoiding walls is necessary to achieve the goal.

Optionally, download all the curriculum documents for Module 15.

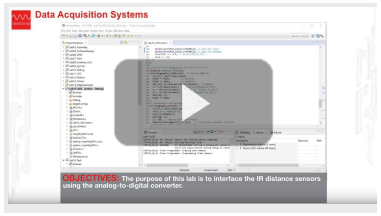
15.1 **TI-RSLK Module 15 - Lecture video part I - Data acquisition systems - Theory**
In this module you will learn how to create a real-time data acquisition system that measures distance from three IR sensors.



15.2 **TI-RSLK Module 15 - Lecture video part II - Data acquisition systems - Performance measurements**
In this module you will learn how to create a real-time data acquisition system that measures distance from three IR sensors.



15.3 **TI-RSLK Module 15 - Lab video 15.1 - Testing IR measurements using the ADC**
The purpose of this lab is to interface IR distance sensors that allow the robot to explore its world.



Module 15

Lab 15: Data Acquisition Systems

Lab: Data Acquisition Systems

15.0 Objectives

The purpose of this lab is to interface IR distance sensors that allow the robot to explore its world, see Figure 1.

1. You will use the ADC to input data into the microcontroller.
2. You will use periodic interrupts to sample the ADC at a regular rate.
3. You will study the noise generated in the data acquisition system.
4. You will evaluate a simple digital filter in an attempt to improve **signal to noise** ratio, which is defined as the signal amplitude divided by the noise amplitude.
5. You will evaluate the accuracy and resolution of the measurement.

Good to Know: You have created a sampling data acquisition system in Labs 6 and 10. I.e., the software sampled the line sensor 100 times/sec. However, in this lab you will study sampling in a more fundamental way. There are three tasks that most embedded systems perform: collecting data, making decisions, and affecting outputs. In this lab, we will focus on collecting data, but the robot challenge will require decision making and outputs.

15.1 Getting Started

15.1.1 Software Starter Projects

Look at these three projects:
ADCSWTrigger (busy-wait ADC interface, simple digital filter, periodic interrupt sampling),
Lab13_Timers (your solution to Lab 13),
Lab15_ADC(starter project for this lab)

15.1.2 Student Resources (in datasheet directory)

MSP432P4xx Technical Reference Manual, ADC14 (SLAU356)
MSP432P401R Datasheet, msp432p401m.pdf (SLAS826)
GP2Y0A21YK0F_IR_Distance_Sensor.pdf, datasheet
Pololu Romi Chassis User's Guide.pdf

15.1.3 Reading Materials

Volume 1 Sections 10.1, 10.4, and 10.5
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Section 8.4, and Chapter 10
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

Good to Know: Analog to digital conversion is one of the most basic operations a microcontroller performs. ADC sampling requires us to make approximations in both the amplitude and time dimensions. Noise is usually the limiting factor on most data acquisition systems.

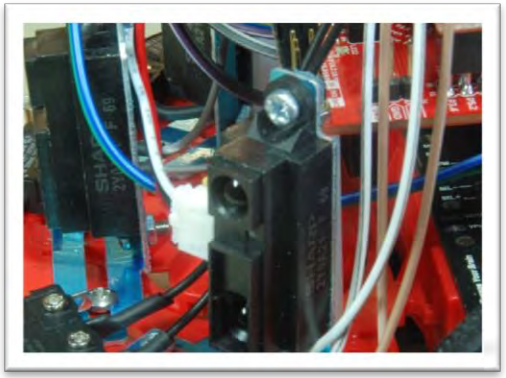


Figure 1. Three IR distance sensors positioned at the front of the robot.

15.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
2	IR bracket pair with 4X bolt and 4X nut	Pololu	#2679
3	IR sensor	Pololu	#136
3	IR cable	Pololu	#1799
3	10 uF tantalum capacitors	AVX	TAP106K020SCS

Table 1 Parts needed for this lab.

Lab: Data Acquisition Systems

15.1.5 Lab equipment needed

Oscilloscope (one or two channels at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling
Optional: Spectrum Analyzer

15.2 System Design Requirements

The first goal is to attach three GP2Y0A21YK0F IR distance sensors to the robot, and then interface the outputs of the sensors to inputs of the ADC converter on your MSP432 Launchpad. You will then convert raw ADC signal to "distance" from the analog domain to the digital domain. Software plus calibration will allow the robot to measure distance to the wall. The range of the measurement will be 70 to 800 mm. The resolution will be about 1 mm at 200 mm.

Figure 2 shows the measured relationship between output of the IR sensor and the distance to the wall (use a block wood and a ruler). Notice the non-monotonic behavior of the sensor... For example, if the system records a sensor value of 2 V, it could mean 33 mm or 130 mm. During the robot challenge you will endeavor to keep the robot away from the wall, so you will assume the sensor distance is greater than 70 mm. Due to the nature of the robot challenges, we are not interested in distances beyond 800 mm.

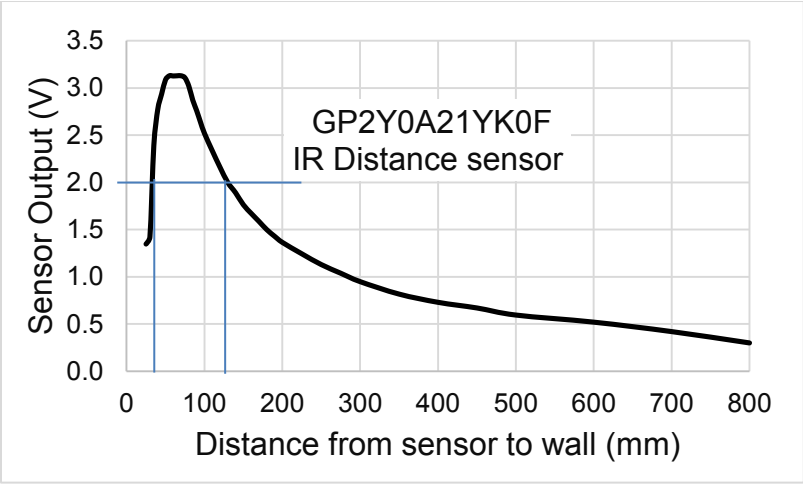


Figure 2. Typical sensor output as a function of distance.

You will develop a function that converts raw ADC samples into distance measured from the wall. Let **n** be a 14-bit sample from the ADC (0 to 16383), and **X** be the distance in mm from the sensor to the wall. The basic form of this nonlinear transfer relation is hyperbolic,

$$\mathbf{X} = A/(\mathbf{n} + B)$$

where A and B are calibration coefficients to be empirically determined.

The above equation defines distance from the sensor to the wall. Your system will however define all distances in mm from a common spot on the robot ($\mathbf{D_r}$, $\mathbf{D_c}$, $\mathbf{D_l}$), as illustrated in Figure 3. This common reference will allow the robot to calculate angle to the wall using geometry and two distance measurements. To handle this change of reference, we will introduce a third calibration coefficient. Define $\mathbf{n_r}$, $\mathbf{n_c}$, and $\mathbf{n_l}$ as the three ADC inputs. Let A_r, A_c, A_l, B_r, B_c, B_l, C_r, C_c, and C_l, be calibration coefficients.

$$\mathbf{D_r} = A_r/(\mathbf{n_r} + B_r) + C_r$$
$$\mathbf{D_c} = A_c/(\mathbf{n_c} + B_c) + C_c$$
$$\mathbf{D_l} = A_l/(\mathbf{n_l} + B_l) + C_l$$

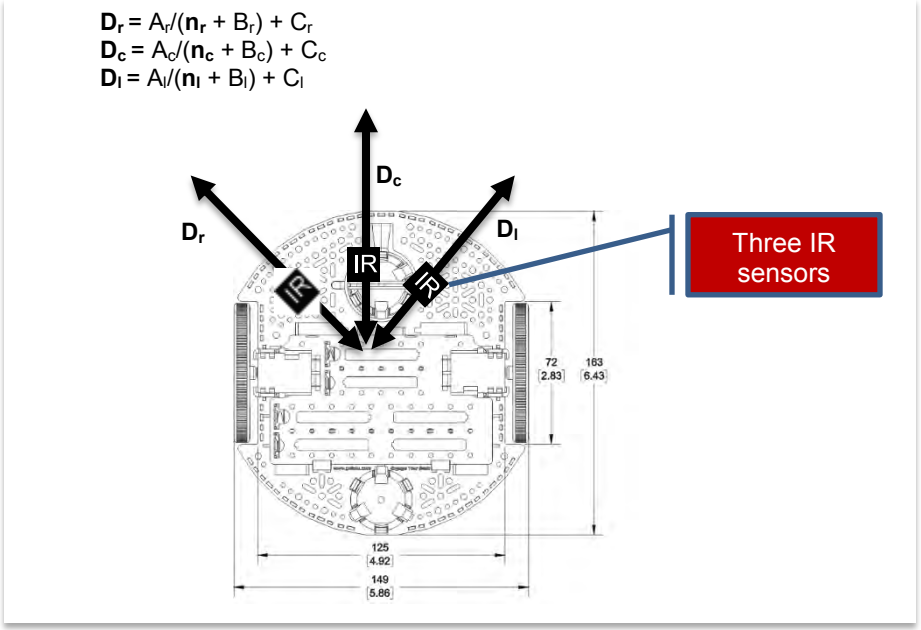


Figure 3. Define distance measured from a central point on the robot.

Lab: Data Acquisition Systems

The maximum measurement distance for the sensor is 800 mm, so if the ADC value is less than a certain threshold, your function should return 800+C. The C prototypes for your functions are

```
int32_t LeftConvert(int32_t nl);       // returns left distance in mm
int32_t CenterConvert(int32_t nc);   // returns center distance in mm
int32_t RightConvert(int32_t nr);      // returns right distance in mm
```

The second goal of this lab is to use periodic interrupts to **sample** the ADC. Triggering the ADC periodically is defined as **sampling**, allowing the system to process the data in both the time and frequency domains. For example, collecting data periodically allows you to implement a digital filter, which passes some data of some frequencies while rejecting others. The purpose of the digital filter is to improve signal to noise ratio. You will study this low pass filter

$$y(n) = (x(n)+x(n-1)+\ldots+x(n-N-1)/N$$

for N = 1 to 512. x(n) is the current sample, x(n-1) is the previous sample, x(n-2) is two sample ago, … and y(n) is the current filter output. You will use the sampled data to study fundamental concepts like the range, resolution, precision, the Oversampling and Nyquist Theorem, aliasing, noise, probability mass function, signal to noise ratio, and the Central Limit Theorem.

An **impulse** digital sequence has one nonzero value and the rest of the points in the sequence are zero, …,0,0,0,1,0,0,0,… The **impulse response** of a filter is the output of the filter given the input is an impulse. If N=4, the impulse response of this filter is …,0,0,0,¼ ,¼,¼,¼,0,0,0,… This filter is called a **finite impulse response** (FIR) filter, because the impulse response has a finite number on nonzero outputs.

A **step** digital sequence has an infinite number of zeros, followed by an infinite number of non-zeros of the same value, …,0,0,0,1,1,1,… The **step response** of a filter is the output of the filter given the input is a step function. If N=4, the step response of this filter is …,0,0,0,0.25 ,0.5,0.75,1,1,1,… In other words if the distance to the wall were to change, this filter will cause a delay in the time the software sees this change in input. You will choose a filter than improves signal to noise ratio without causing too much delay in the step response.

The third goal is to evaluate the accuracy of the distance measurement. **Accuracy** is defined as the difference between truth and measured. Let x_t be the true distance from the robot reference point to the wall, as measured with a ruler. The instrument accuracy is the absolute error referenced to the National Institute of Standards and Technology (NIST) of the entire system including transducer, electronics, and software. Let x_m be the values as measured by the instrument. We define average accuracy of full scale in percent as

$$\frac{100}{n}\sum_{i=0}^{n}\frac{|x_{ti}-x_{mi}|}{x_{t\max}}$$

*The system **resolution** is the smallest input signal difference, Δx that can be detected by the entire system including transducer, electronics, and software. The resolution of the system is limited by noise processes in the transducer itself, noise processes in the electronics, and the number of bits in the ADC. For this lab, resolution will be limited by noise in the IR distance sensor.*

15.3 Experiment set-up

You will attach three IR distance sensors near the front of the robot. Recall that the sensor is confused for distances from 0 to 70 mm (Figure 2), it is preferable to recess the sensors away from the edge of the robot as much as possible. The robot in Figure 1 has the sensors recessed 25 mm from the point at which the bump sensors (used in Module 10) will activate. Therefore, this robot will report incorrect distances when 25 to 70 mm from a wall. The exact placement and angle will be readjusted as part of the robot challenge you attempt. For this lab, it is not critical exactly where the sensors are placed, see Figure 3.

The interface circuit in Figure 4 connects the sensor output directly to the ADC input. Any ADC input pins could have been used, but the issue with a complex design is assigning pins for special purpose like timer PWM, edge-triggered input, UART, SPI and ADC. The three pins shown in Figure 4 do not conflict with the other labs in these modules. Connect the IR sensor to +5V supply, which on the Pololu Motor Driver Power Distribution board labeled VREG. On the LaunchPad, +5V power is simply labeled 5V. The sensor is very noisy, and you will need a supply capacitor for each sensor. The figure shows 10 µF, but any capacitor from 4.7 to 47 µF would be ok.

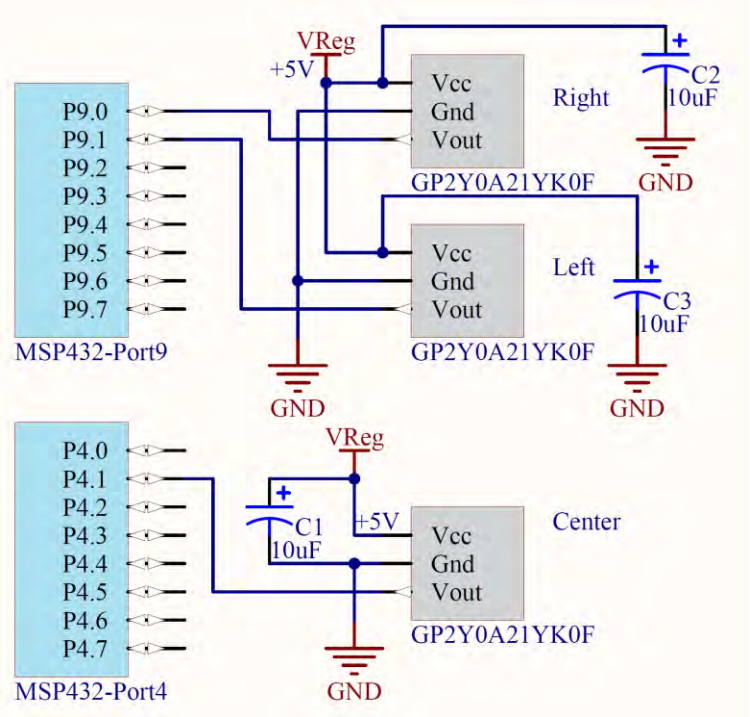


Figure 4. One possible interface circuit for the IR sensors.

Even though the sensor is powered with 5 V, the output will only range from 0 to 3.1 V, see Figure 2. Therefore, it is safe to connect this signal to the MSP432 powered at 3.3 V.

Note: You must use the 0 to 3.3V ADC range and not use the precision internal voltage reference at 2.5 V.

As an option, you could build three analog low pass filters (LPF) and place them between the sensor and ADC. If the sampling frequency is 1000 Hz, then the cutoff for the analog LPF should be about 100 Hz. For this you will need op amps to design a LPF filter and will need a breadboard, the components have not been included in the lab.

15.4 System Development Plan

15.4.1 Explore the ADC, discover the Central Limit Theorem

In this section we will explore how the ADC works using TI's Launchpad. You will using a power supply. Connect a voltage **of about 3 V (any value from 2.5 to 3.1V)** to P4.7. P4.7 is ADC channel 6. Build, debug and run the project **ADCSWTrigger**. This project samples P4.7 at 1000 Hz using SysTick interrupts, implements a simple averaging digital filter, and performs statistical analysis on the collected data. In particular, it calculates a PMF (probability distribution of noise), mean (μ), range (max-min), variance (σ^2) and standard deviation (σ). If you run a terminal emulator like PuTTY or TExaSdisplay, you can see the output of this statistical analysis.

Note: Alternatively, if you have interfaced the LCD to your robot, then outputting these parameters to the LCD will simplify testing.

Place a voltmeter on the P4.7 input and observe the true voltage. Comparing the true voltage with the ADC digital output allows you to see how the ADC operates. Assuming there were no noise and the input were constant, you would expect all the ADC samples to be equal. The fact that the samples are not the same is the result of **noise**. Without noise, the variance and standard deviations would be zero. The **coefficient of variation** (CV) is the standard deviation divided by the mean (μ),

$$CV = \sigma/\mu$$

1/CV is a simple estimate of the signal to noise ratio (SNR). With the input voltage around 3 V, we will approximate the precision in bits of the system as

$$\mathbf{\log_2(\mu/\sigma)}$$

This project also implements a simple digital filter. This filter calculates the average of the last N samples.

$$y(n) = (x(n)+x(n-1)+\ldots+x(n-N-1)/N$$

Averaging is a simple method to improve signal to noise ratio. In this project the value N is defined in the variable Size, varying from N = 1 to 512. By pressing and releasing either of the LaunchPad switches, the software cycles through these ten values of N. With the input voltage at about 3V, collect 1/CV (SNR) and $\log_2(\mu/\sigma)$ data as a function of N = 1, 2, 4, …, and 512. What do you observe?

Next review the Central Limit Theorem (CLT) from your Probability and Statistics class. Look up the assumptions to see if the CLT applies to these measurements. The CLT states that as independent random variables are added, their sum tends toward a Normal or Gaussian distribution. In this experiment you should find, as N is increased the PMF goes from having multiple peaks to having just one peak (see lecture slides). This behavior will be even more pronounced when studying the noise from the IR distance sensor.

15.4.2 Study the digital filters (optional)

If you wish to learn more about the simple averaging filter, open the spreadsheet **FIR_Digital_LowPassFilter.xls** (you should have this in your folder when you opened up the zip file). You can change the sampling rate (fs) and filter size (N) and visualize the frequency and step responses. The two parameters you can adjust are highlighted in yellow. If the sampling rate is 2000 Hz, and the size N is 64, then the filter has a cutoff frequency (fc) of 16 Hz, see Figure 5.

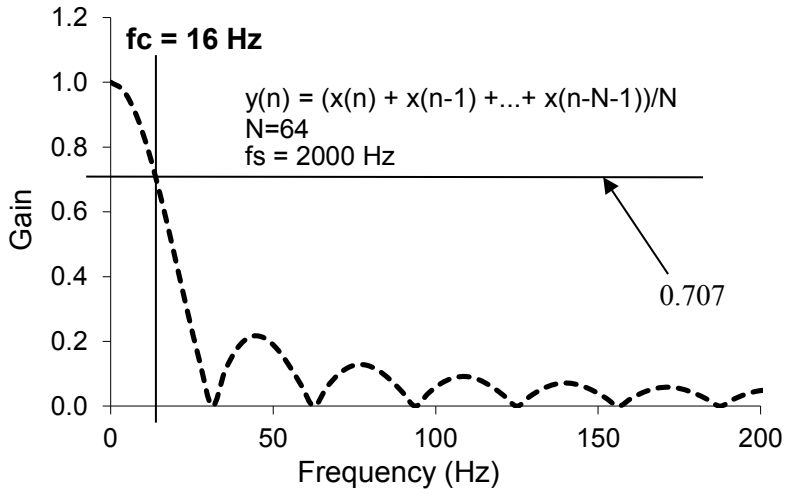


Figure 5. Frequency response of averaging filter with N=64.

15.4.3 Low-level ADC software driver

The first software step is to write two functions that sample the analog signal from the center IR distance sensor. The prototypes for these functions are

void ADC0_InitSWTriggerCh12(void); // initialize P4.1, channel A12
uint32_t ADC_In12(void); // sample P4.1, channel A12

Basically you will convert the initialization and sample function for channel 6 (P4.7) to channel 12 (P4.1), leaving all the design decisions the same, configured for the following:

- Single channel
- Software start
- Busy wait synchronization
- 14-bit, unsigned binary
- 3.3V V(R+), analog input range is 0 to 3.3V
- ADC14MEM0 address

Note: You will not be able to complete this lab without reading the MSP432 data sheet. Look at the chapter on ADC14, and go line by line through the existing ADC0_InitSWTriggerCh6 and ADC_In6 functions. These two functions work, but you need to understand each line, by looking up each of the registers it accesses. Once you understand each line, you will be able to convert it from sampling channel 6 to sampling channel 12.

To test these functions you can run Program15_1. The ISR samples channel 12, runs an averaging digital low pass filter, and passes the data through a mailbox (variable and semaphore) to another thread.

```
void Program15_1_ISR(void){  // runs at 2000 Hz
  uint32_t RawADC;
  P1OUT ^= 0x01;                    // profile
  P1OUT ^= 0x01;                    // profile
  RawADC = ADC_In12();  // sample P4.1/channel 12
  ADCvalue = LPF_Calc(RawADC);
  ADCflag = 1;                      // semaphore
  P1OUT ^= 0x01;                    // profile
}
```

15.4.4 Signal to Noise Ratio of IR distance sensor

To analyze sensor noise, you can use **Program15_1**, which is similar to the project **ADCSWTrigger**, replacing all the channel 6 accesses to channel 12. Run **Program15_1** and determine the SNR for various sizes of the averaging filter, from **N = 32 to 512.**

Low pass FIR $y(n) = (x(n)+x(n-1) +\ldots+x(n-N-1)/N$

While deciding the size of the averaging, also consider the step response. In other words, let the input go from 0 to 1, and calculate the output of filter as a function of time, assuming a sampling rate of 2000 Hz. Calculate the time constant of the filter, defined as the time it takes the output to achieve 63% ($1-e^{-1}$) of the new value given a change in input. The time constant of the FIR filter is 20 ms, see Figure 6. Select a value of N for your robot.

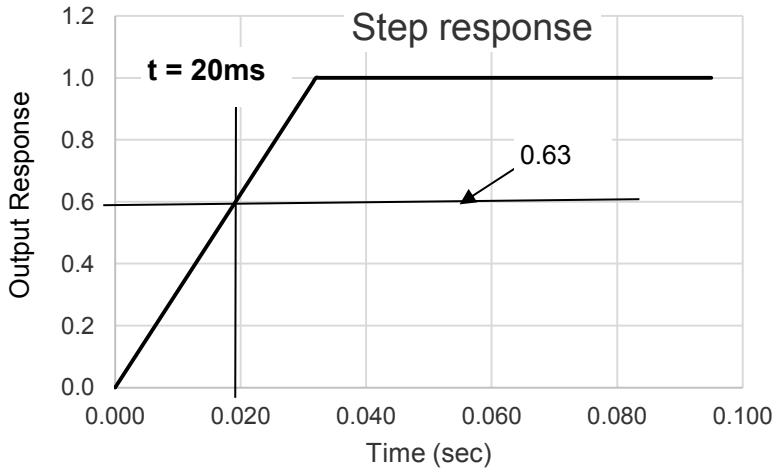


Figure 6. Step response of averaging filter with N= 64.

Note: Since the time constant of the motor is about 100ms, we wish to have the time constant of the filter to be smaller than 100ms.

If you have access to a spectrum analyzer, it is very interesting to observe the noise in the frequency domain. Figure 7 illustrates there is significant noise throughout the frequencies, with peaks at multiples of 1 kHz. The output of a spectrum analyzer is given in decibels full scale. The spectrum analyzer in Figure 7 has a full scale of 5V, so

$$dB_{FS} = 20 \log_{10}(V/5)$$

The noise at 1 kHz, -35 dB, is the equivalent of about 90 mV. On the MSP432, a 90 mV noise will reduce the precision of the system from 14 bits possible with the ADC to $\log_2(3.3V/0.09V)$, which is about 5 bits. The purpose of the digital filter is to remove some of the noise, and improve precision.

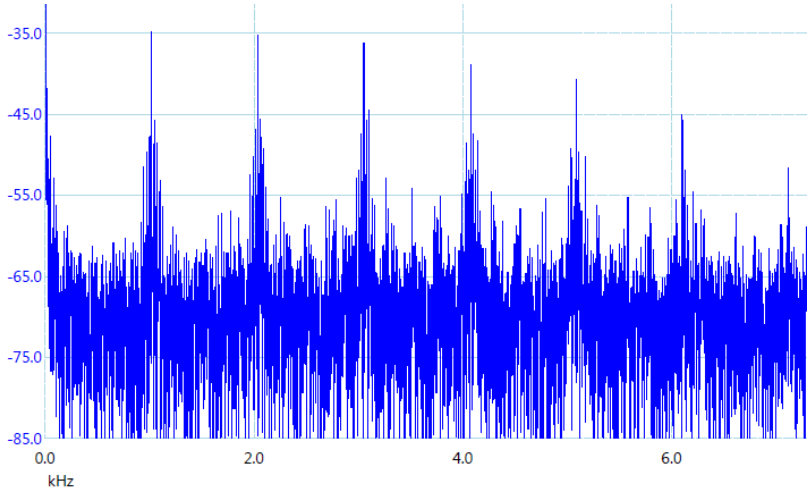


Figure 7. Frequency spectrum of the output of the distance sensor (without analog filter).

Lab: Data Acquisition Systems

15.4.5 Three-channel ADC software driver

The next software step is to write two functions that sample all three analog signals from the sensors. You will need to make three copies of the digital low pass filter, so all channels are filtered. The prototypes for your functions are

void ADC0_InitSWTriggerCh17_12_16 (void);
void ADC_In17_12_16(uint32_t *ch17, uint32_t *ch12, uint32_t *ch16);
Basically you will convert the initialization and function for channels 6 and 7 to channel 12 (P4.1), leaving most the design decisions the same

- Three channel sampling
- Software start
- Busy wait synchronization
- 14-bit, unsigned binary
- 3.3V V(R+), analog input range is 0 to 3.3V
- ADC14MEM2, ADC14MEM3, ADC14MEM4 addresses

To test these functions you can run Program15_2. The ISR samples the three channels, runs three averaging digital low pass filters, and passes the data through a mailbox (variables and semaphore) to another thread.

Note: Again, you must read the MSP432 data sheet when looking at the existing ADC0_InitSWTriggerCh67 and ADC_In67 functions. These two functions work, but you need to understand each line, by looking up each of the registers it accesses. Once you understand each line, you will be able to convert it from sampling channels 6 and 7 to sampling channels 17, 12, and 16.

```
volatile uint32_t nr,nc,nl;
void Program15_2_ISR(void){  // runs at 2000 Hz
  uint32_t raw17,raw12,raw16;
  P1OUT ^= 0x01;          // profile
  P1OUT ^= 0x01;          // profile
  ADC_In17_12_16(&raw17,&raw12,&raw16);  // sample
  nr = LPF_Calc(raw17);  // right is channel 17 P9.0
  nc = LPF_Calc2(raw12); // center is channel 12, P4.1
  nl = LPF_Calc3(raw16); // left is channel 16, P9.1
  ADCflag = 1;           // semaphore
  P1OUT ^= 0x01;          // profile
}
int Program15_2(void){ // example program 15.2
  uint32_t raw17,raw12,raw16; int32_t n; uint32_t s;
  Clock_Init48MHz();
  ADCflag = 0; s = 256; // replace with your choice
  ADC0_InitSWTriggerCh17_12_16();   // initialize
  ADC_In17_12_16(&raw17,&raw12,&raw16);  // sample
  LPF_Init(raw17,s);      // P9.0/channel 17
  LPF_Init2(raw12,s);     // P4.1/channel 12
  LPF_Init3(raw16,s);     // P9.1/channel 16
  UART0_Init();           // initialize UART0
  LaunchPad_Init();
  TimerA1_Init(&Program15_2_ISR,250);  // 2000 Hz
  UART0_OutString("Program 15.2 \n");
  EnableInterrupts();
  while(1){
   for(n=0; n<2000; n++){
     while(ADCflag == 0){};
     ADCflag = 0; // show every 2000th point
   }
   UART0_OutUDec5(nl);UART0_OutUDec5(LeftConvert(nl));
   UART0_OutUDec5(nc);UART0_OutUDec5(CenterConvert(nc));
   UART0_OutUDec5(nr);UART0_OutUDec5(RightConvert(nr));
   UART0_OutChar('\n'); // once a second
  }
}
```

Lab: Data Acquisition Systems

Connect a scope to P1.0 and measure the time between interrupts and the time within the ISR. Figure 8 shows this system (Size=256) executes 25 µs each time in the ISR. At 2 kHz, the ADC sampling and digital filtering consume 25/500 =0.05 (5%) of the processor time.

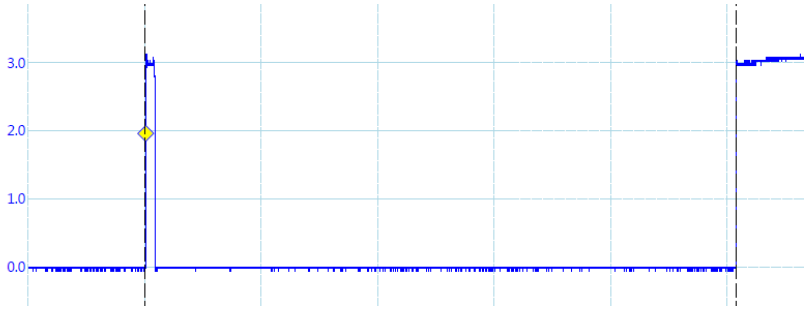


Figure 8. Thread profile. Scope attached to P1.0. The time scale is 5 µs per division.

15.4.6 Calibration and accuracy

Implement the three conversion functions, and use **Program15_2** first to calibrate and then to test these functions. Define exactly where on the robot you wish to set the reference point, see Figure 3. Use a ruler to measure the true distances ($\mathbf{D_r}$, $\mathbf{D_c}$, $\mathbf{D_l}$), and use **Program 15_2** to display the raw ADC values ($\mathbf{n_r}$, $\mathbf{n_c}$, and $\mathbf{n_l}$). Collect about 10 points for each sensor and fit the data to find the calibration coefficients for each sensor. Limit your calibration for distances between than 70 mm and 800 mm from the sensor.

Enter the coefficients into your software, and repeat the process collecting another set of 10 measurements for each sensor. Calculate the average accuracy of full scale in percent for each sensor.

15.4.7 Discovering the Nyquist Theorem

The **Nyquist Theorem** states that if the signal is sampled with a frequency of f_s, then the digital samples only contain frequency components from 0 to ½ f_s. Conversely, if the analog signal does contain frequency components larger than ½ f_s, then there will be an aliasing error during the sampling process. Aliasing is when the digital signal appears to have a different frequency than the original analog signal.

Although the ADC is sampled at 2000 Hz, since Program15_2 outputs once a second, the data observed on the terminal program can be considered to have been sampled at 1 Hz.

1) Observing one sensor output, oscillate the wall (block of wood) 100 to 200 mm from the sensor with a period of 4 to 10 seconds. Notice data tracks the signal in such a manner that you could reconstruct both the frequency and amplitude of the oscillations.

2) Very carefully attempt to oscillate the wall at a constant amplitude but with a frequency of 0.5 Hz (period of 2 sec). At this frequency the sampled data will oscillate 100,200,100,200,100… This is the Nyquist frequency (½ f_s) at which the system transitions from operational region (able to recover amplitude and frequency) to a region at which the digital data cannot be used to recover the amplitude and frequency of the oscillations.

3) Finally, oscillate the wall at a constant amplitude but with a frequency much faster than 0.5 Hz. Data existing at frequencies above ½ f_s will be aliased (frequency folded into 0 to 0.5Hz), and the digital data cannot be used to recover the amplitude and frequency of the oscillations. The problem with aliasing is that high frequency noise will appear as low frequency signals. Therefore we must remove high amplitude signals at or above ½ $\mathbf{f_s}$ using an analog low pass filter.

15.5 Troubleshooting

IR distance sensors don't work:

- Check the wiring as described in Figure 4,
- Using a voltmeter observe power (+5V), ground, and signal directly on the sensor

ADC doesn't work:

- Run the **ADCSWTrigger**. This project should properly sample channel 6
- Review the data sheet and double check each register accessed by your software

ISR takes more than 25us to execute:

- Notice that this implementation of the filter requires about 9us for one channel and 25 us for three channels. This is because each filter calculation requires one subtraction, one addition and one division.
- Make sure there are no loops in the ISR (except for the busy-wait in the ADC sampling).

Statistical calculations are incorrect:

- Look at the collected data in the array Data[N]. Bad statistics may be a result of bad data.
- You can reduce the statistical array to N=8, and perform the statistical calculated by hand to check the software.
- 100*Sum2 can overflow if the data is very noisy.

15.6 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.

- What is the mathematic relationship between ADC input voltage and digital output number?
- What is the limiting factor in this system that restricts resolution and accuracy of the distance measurement?
- Why did the function **ADC_In17_12_16** use call by reference parameter passing?
- What happens as the size of the averaging filter is doubled?
- Why are interrupts required in this lab? i.e., what do interrupts enable us to do?
- How is the mailbox used in the lab? What does ADCflag=0 mean? What does ADCflag=1 mean?
- What would it mean if the ADCflag were already 1 at the time the ISR is trying to set it to 1 again?

15.7 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,

- If you add three analog filters between the sensor output and ADC input (fc=100 Hz) you can greatly reduce the noise and could also reduce the sampling frequency and size of the digital filter.
- A median filter is an alternate digital filter than could be added to improve signal to noise ratio.
- If you performed Lab 11 (LCD), then you could output data to the LCD, making it easier to debug, calibrate, and test.

15.8 Which modules are next?

This was our first of many uses of interrupts in this course. The following modules will build on this module:

Module 16) Interface tachometers to the microcontroller and use input capture to measure wheel velocity.
Module 17) Combine modules 12, 13 and 16 to develop closed loop motor controllers. In this module you will be able to spin the motors at a constant speed.

15.9 Things you should have learned

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

- Use periodic interrupts to implement sampling
- Use the ADC to convert from analog to digital domain
- Noise is difficult and important problem to solve; with a constant voltage connected the LaunchPad, noise will limit the 14-bit ADC to 10 or 11 bits; noise is often the limiting factor for resolution and not the number of bits in the ADC
- Software can efficiently and effectively implement filtering
- Software can effectively handle a nonlinear (hyperbolic) transducer
- Accuracy depends on two processes: resolution and calibration. Resolution depends on noise, and calibration depends on the stability of the transducer.

Module 16

Introduction: Tachometer

Introduction: Tachometer

Educational Objectives:

UNDERSTAND Timers measuring period
INTERFACE the tachometer
CREATE A low-level software driver to implement input capture
DESIGN A system that can measure wheel rotational speed

Prerequisites (Modules 10, 12, 13)
- Periodic interrupts using SysTick (Module 10)
- Mechanical and electrical interfaces of motors (Module 12)
- Timer_A periodic interrupts (Module 13)

Recommended reading materials for students:
- Volume 1 Sections 4.1, 9.4, and 9.7
  Embedded Systems: Introduction to the MSP432 Microcontroller, ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or

- Volume 2 Sections 2.2, 5.4, and 6.1
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

We will combine this module, together with Modules 12, 13, and 17, to create a closed-loop control system. With a control system, we can independently set the rotational speed of each motor. The control system measures speed and uses feedback to adjust the PWM duty cycle of each motor to achieve the desired speed. With a control system the robot can move in a straight line, run at a desired speed, travel a prescribed distance, or turn a prescribed angle.

A tachometer is a sensor with digital outputs that relate to rotational speed. The Romi Shaft encoder (Pololu # 3542) has two outputs, and each output pulses 360 times per rotation, see Figure 1. If we measure the **Period** (in sec) of one of the tachometer signals, we can calculate the motor **Speed** in rpm as

Speed = 360*60/**Period**

We use input capture mode to make time measurements on input signals. The MSP432 microcontroller has four **General Purpose Timer Modules** called Timer_A. Each timer has one 16-bit timer and seven associated capture/compare registers. Similar to the ADC measurements in Module 15, we are concerned with range, resolution, precision, noise, and accuracy.

In this lab, you will use Timer A3 to create two icapture inputs for the tachometer interfaces. There will be an interrupt on each rising edge, and the timer will measure the periods of the two inputs.

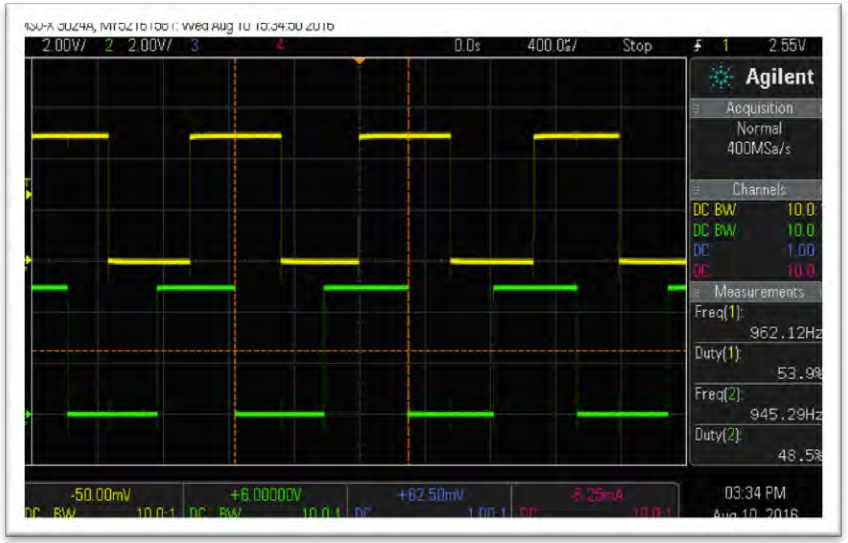


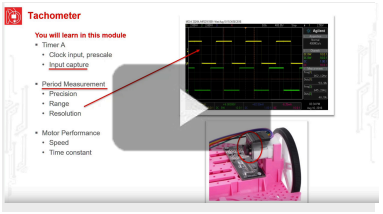
*Figure 1. Scope trace of the two outputs of the tachometer, period*360 is the time for one revolution. (From https://www.pololu.com/product/3542/)*

16. TI-RSLK Module 16 – Tachometer

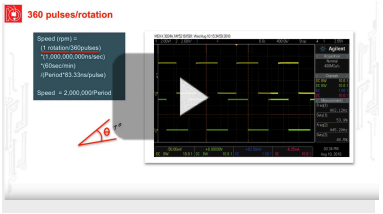
In this module, you will learn how to interface the tachometers that enable the robot to measure motor rotational speed. Tachometer data allows your software to drive straight, drive for a prescribed amount of distance or turn at a prescribed angle.

Optionally, download all the curriculum documents for Module 16.

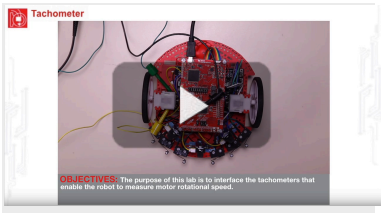
16.1 **TI-RSLK Module 16 - Lecture video part I - Tachometer - Input capture**
In this module you will learn how to design a system that can measure wheel rotational speed.



16.2 **TI-RSLK Module 16 - Lecture video part II - Tachometer - Interface**
In this module you will learn how to design a system that can measure wheel rotational speed.



16.3 **TI-RSLK Module 16 - Lab video 16.1 - Testing the Tachometer**
You will learn encoder, motor speed. motor direction, motor Performance, speed, time constant.



Module 16

Lab 16: Tachometer

Lab: Tachometer

16.0 Objectives

The purpose of this lab is to develop the software needed to measure motor speed. In this module,

1. You will learn more about the MSP432 Timer_A module.
2. You will configure Timer A3 for input capture measurements.
3. You will develop low-level software drivers to measure distance and speed of the two motors on the robot.

Good to Know: A typical application for embedded systems is control. Sensors measure the state of the system (motor speed), and software adjusts the actuator (PWM to motors) in an attempt to control the system in a desired manner (constant speed).

16.1 Getting Started

16.1.1 Software Starter Projects

Look at these two projects:
PeriodMeasure (uses a timer A0 to measure period on P7.3)
Lab16_Tach (starter project for this lab)

Note: You will not be able to run the **PeriodMeasure** project on the robot because this project uses Timer A0, and you are using Timer A0 for the robot's PWM outputs. You will use Timer A3 for the tachometer. Timers A1 and A2 are free to use as periodic interrupts.

16.1.2 Student Resources (in datasheet directory)

MSP432P4xx Technical Reference Manual, Timer_A (SLAU356)
MSP432P401R Datasheet, msp432p401m.pdf (SLAS826)

MotorDriverPowerDistribution.pdf	Data sheet for power board
Pololu Romi Chassis User's Guide.pdf	How to build the robot

16.1.3 Reading Materials

Volume 1 Sections 4.1, 9.4, and 9.7
"Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 2.2, 5.4, and 6.1
"Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

16.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	Romi Chassis Kit - Red	Pololu	3502
1	Motor Driver and Power Distribution Board for Romi	Pololu	3543
1	Romi Encoder Pair Kit, 12 CPR	Pololu	3542
2	Rechargeable Battery, Pack of 4, Metal Hydride 1300 mAh, 1.2V, AA	Energizer	626831
4	1.375in 4-40 Nylon standoff	Keystone	4809
2	0.187in 4-40 metal nut	Keystone	4694
6	0.5in 4-40 Nylon machine screw	Pololu	1962

Lab: Tachometer

16.1.5 Lab equipment needed

Oscilloscope (one or two channels at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling

Warning: Disconnect the VREG↔+5V wire when the LaunchPad USB cable is connected to the PC. Connect the VREG↔+5V wire when the robot is running on battery power. This way the motors always get power from the batteries, and never get power from the USB.

16.2 System Design Requirements

The first goal of this lab is to write Timer_A software that can measure period from the two encoders. The counter of Timer_A is 16 bits wide, so the period measurement will have a **precision** of 16 bits. This means you can measure about 65536 different periods. The **resolution** is defined as the smallest change in period that the measurement can distinguish. The resolution in input capture mode is equal to the period of the selected clock. If you choose the SMCLK at 12 MHz and a prescale of 1, the period measurement resolution will be 83.33 ns. The maximum period that can be measured is the precision in alternatives times the resolution. At this clock and prescale, the **maximum** period that can be measured is about 5.4 ms.

The second goal is the use the period to determine motor speed. Since there are 360 pulses per rotation, this 5.4-ms maximum means the slowest motor speed that can be measured will be about 30 rpm. If **Period** is the period in 83.33-ns units, then the **Speed** in rpm can be calculated as

Speed (rpm) = (rotation/360pulses)*(1,000,000,000ns/sec)
*(60sec/min)/(**Period***83.33ns/pulse)

or

Speed = 2,000,000/**Period**

The third goal is to use the second input of the encoder to determine which direction the motor is spinning. You will write software that counts the number of pulses observed on each wheel as the robot moves. You will add to a counter as the robot moves forward, and you will subtract from a counter as the robot moves backward.

16.3 Experiment set-up

Refer to the data sheets of the MDPDB and encoder to see how to connect the motors and encoders. See Figure 1. Detailed directions can be found at

https://www.pololu.com/docs/0J68/all

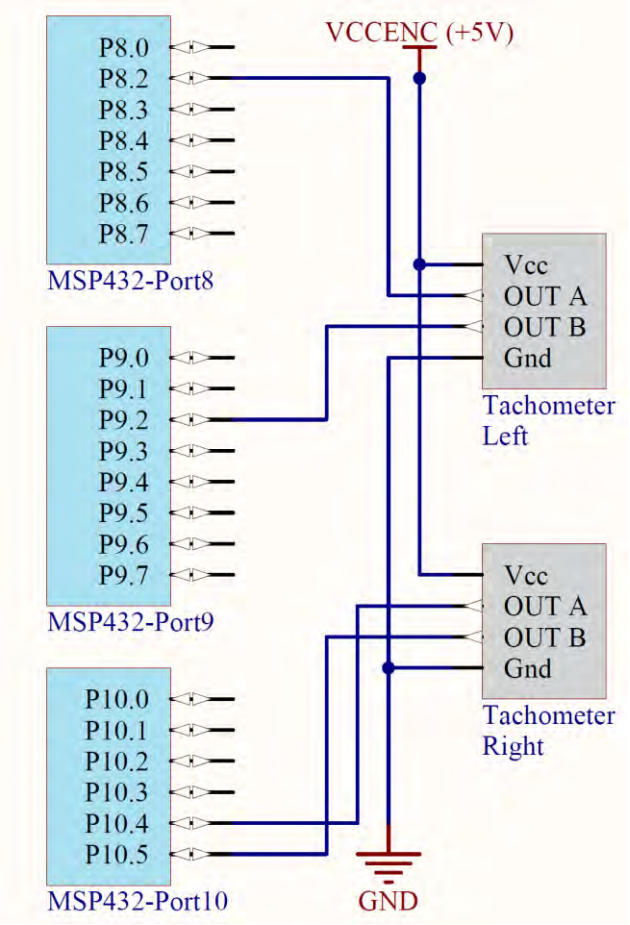


Figure 1. Possible interface for connecting the encoders to the MSP432.

Lab: Tachometer

LaunchPad	MDPDB	Encoder	Description
P8.2/TA3CCP2	ELA	OUT A	Left Encoder A
P9.2/GPIO	ELB	OUT B	Left Encoder B
P10.4/TA3CCP0	ERA	OUT A	Right Encoder A
P10.5/GPIO	ERB	OUT B	Right Encoder B

16.4 System Development Plan

16.4.1 Study the existing input capture

An efficient mechanism for learning a new skill is to first study existing art. The project **PeriodMeasure** will measure the period on P7.3 using Timer A0. You can connect a 0 to 3.3V digital wave to P7.3 using a signal generator, or you can use this main program to create a test wave. To use this program you will need to connect P2.4 output to the P7.3 input.

```
void PeriodMeasure(uint16_t time){
  P2_0 = P2_0^0x01;           // thread profile, P2.0
  Period = (time - First)&0xFFFF; // 16 bits, 83.3 ns
  First = time;                 // setup for next
  Done = 1;
}
#define PERIOD 1000  // must be even
// connect P2.4 output to P7.3
// creates a PERIOD (us) wave out P2.4
int main(void){
  Clock_Init48MHz();  // 48 MHz; 12 MHz Timer A clock
  First = 0;          // first will be wrong
  Done = 0;           // set on subsequent
  TimerA0Capture_Init(&PeriodMeasure);// capture mode
  P2->SEL0 &= ~0x11;
  P2->SEL1 &= ~0x11;  // configure P2.0 and P2.4 as GPIO
  P2->DIR |= 0x11;    // P2.0 and P2.4 outputs
  EnableInterrupts();
  while(1){
    P2_4 ^= 0x01;     // create output
    Clock_Delay1us(PERIOD/2);
  }
}
```

The resolution of the measurement is 1/12MHz = 83.33 ns and the range is about 10 us to 5.44 ms. If the period is 1 ms, then the software will return a result of 12000. This example uses **bit-banding** to access Port 2 in order to eliminate the critical section caused by the read-modify-write access to the shared global (P2->OUT).

Note: You will not be able to complete this lab without reading the MSP432 data sheet. Look at the chapter on Timer_A, and go line by line through the existing **TimerA1_Init** and **TA1_0_IRQHandler** functions within the **PeriodMeasure** project. This measurement works, but you need to understand each line, by looking up each of the registers it accesses. Once you understand each line, you will be able to convert it from measuring on P7.3 using Timer A0 to measuring both P10.4 and P8.2 using Timer A3.

16.4.2 Low-level software driver

Write the low-level driver to handle input capture on P10.4 and P8.2 using Timer A3. The prototype for the low-level driver is:

```
void TimerA3Capture_Init(void(*task0)(uint16_t time),
     void(*task2)(uint16_t time));
```

This is an example of a vectored interrupt. The rising edge of P10.4 will cause an interrupt on **TA3_0_IRQHandler**, and the rising edge of P8.2 will cause an interrupt on **TA3_N_IRQHandler**. The **TA3_0_IRQHandler** ISR will call the user function passed in via the **task0** parameter, and the **TA3_N_IRQHandler** ISR will call the user function passed in via the **task2** parameter. The captured time of the edge is passed from the ISR to the user function in a manner similar to the **PeriodMeasure** project. You can use Program16_1 to test the low-level driver. Place the robot on blocks so the wheels do not touch the ground while performing initial testing.

```
uint16_t Period0;   // (1/SMCLK) units = 83.3 ns units
uint16_t First0;    // Timer A3 first edge, P10.4
int Done0;          // set each rising
void PeriodMeasure0(uint16_t time){
  P2_0 = P2_0^0x01; // thread profile, P2.0
  Period0 = (time-First0)&0xFFFF; // 16 bits, 83.3 ns
  First0 = time;    // setup for next
  Done0 = 1;
}
uint16_t Period2;   // (1/SMCLK) units = 83.3 ns units
uint16_t First2;    // Timer A3 first edge, P8.2
int Done2;          // set each rising

  P2_4 = P2_4^0x01; // thread profile, P2.4
  Period2 = (time-First2)&0xFFFF; // 16 bits, 83.3 ns
  First2 = time;    // setup for next
  Done2 = 1;
}
int Program16_1(void){
  Clock_Init48MHz();   // 48 MHz; 12 MHz Timer A
  P2->SEL0 &= ~0x11;
  P2->SEL1 &= ~0x11;   // P2.0 and P2.4 as GPIO
  P2->DIR |= 0x11;     // P2.0 and P2.4 outputs
  First0 = First2 = 0; // first will be wrong
  Done0 = Done2 = 0;   // set on subsequent
  Motor_Init();        // activate Lab 13 software
  TimerA3Capture_Init(&PeriodMeasure0,&PeriodMeasure2);
  Motor_Forward(7500,7500); // 50%
  EnableInterrupts();
  while(1){
    WaitForInterrupt();
  }
}
```

Note: Feel free to modify any of the details of how it works, as long as the overall system can measure motor speed for both wheels.

Adjust the period measurement resolution so that the system can measure period for a range of motor duty cycles from 25 to 100%

16.4.3 Mid-level software driver

Write the software to convert the period measurements into motor speed in rpm. Perform a static motor test while the robot is still on the blocks. For duty cycles {25, 50, 75, and 100%}, measure the motor speed of each motor in RPM.

Write a test program that periodically collects motor speeds versus time using a 100 Hz periodic interrupt. Include the bumper driver from Lab 10 or Lab 14 so the robot stops on a collision. Dump power (duty cycle) and speed data into buffers similar to Lab 10. For very long tests, you can dump into flash ROM. For shorter tests, you can dump into RAM. In the main program, perform these steps running the robot for 10 seconds.

1. Run forward at 25% duty cycle for 2 seconds
2. Run forward at 50% duty cycle for 2 seconds
3. Run forward at 75% duty cycle for 2 seconds
4. Run forward at 100% duty cycle for 2 seconds
5. Run forward at 25% duty cycle for 2 seconds
6. Stop the motors and stop the recording

Run this motor test on blocks and on a flat surface. We define the **time constant**, τ, of the motor as the time it takes to achieve $(1\text{-}e^{-1}) = 0.63$ of the final speed, given a step change in power to the motor. Fit the speed versus time data to an exponential to estimate the time-constant of your motors.

$$y(t) = S_0 + \Delta S\, e^{-t/\tau}$$

where S_0, ΔS, and τ are least squares fit of the y(t) data verses time. Initial time is defined at the point the duty cycle was changed.

16.4.4 High-level software driver

Extend the measurement to initialize the other two input pins. Create two global signed 32-bit counters, one for each motor. In addition to measuring period and motor speed, count the number of edges on each encoder. On each edge add one if moving forward and subtract one if moving backward.

Lab: Tachometer

16.5 Troubleshooting

Input capture interrupts do not occur:

- Check to see if the edges are occurring on P8.2 and P10.4
- Check to see if the trigger flags are being set. Bit 0 of the register TIMER_A3->CCTL[0] should be set by edge of P10.4, and bit 0 of the register TIMER_A3->CCTL[2] should be set by edge of P8.4.
- Check to see if the arm bits are set in Timer A3. Bit 4 of the register TIMER_A3->CCTL[0] arms P10.4, and bit 4 of the register TIMER_A3->CCTL[2] arms P8.4.
- Check to see if the enable bits are set in the NVIC for Timer A3. Bit 14 of the register NVIC->ISER[0] enables T3_0 (P10.4) and bit 15 enables T3_N (P8.2).
- Check to see if the I-bit in the processor is clear.

Interrupts occur over and over:

- Check the hardware with a scope or logic analyzer to make sure the sensor is operating properly
- Make sure you clear the trigger flag (acknowledge) in the ISR. Bit 0 of the register TIMER_A3->CCTL[0] should be cleared by software in the ISR of P10.4, and bit 0 of the register TIMER_A3->CCTL[2] should be cleared by software in the ISR for P8.4

16.6 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. The goal of this module is for you to understand Timer_A and its use for measuring period.

- What does the prescaler do for Timer_A? Why is the prescaler important (i.e., what happens when you change the prescale?)
- What is the precision of the period measurement mean and how is it determined?
- What happens if the motor spins too slowly, e.g., less than 30 RPM?
- What happens if the motor stops, e.g., does not spin at all?
- How do we debug this system if the robot is moving along the ground?
- Why is the time constant of the motor differ if the robot is on blocks versus on the ground?

16.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. For example,

- If you completed Lab 11, add LCD outputs for each of the test functions. Remember to perform LCD output only in the main program and not during an ISR.
- Add software to detect if the motor has stopped or moving less than 30 PRM. Deploy a periodic interrupt that counts the time with the semaphore clear. If 10ms has elapsed and the semaphore is still clear, you can assume the motor is moving slowly or has stopped.
- You could configure the measurement to interrupt on rising and falling edges of all four encoder pins. For each encoder define period as the time from one edge to the next edge, see Figure 1. This means there will be 4*360 (1440) edges per one rotation. In this approach, there are four times as many interrupts. This results in four times the resolution and four times the rate at which measurements are obtained. With the SMCLK at 12 MHz and prescale at 1, the maximum time that can be measured is still 5.4 ms. Consequently, this means the slowest motor speed that can be measured will be about 7.5 rpm.
- If you consider how the speed measurement will be used, you will find a new speed measurement will be needed every 10 ms. During this 10-ms time, there could be multiple input capture events. If the data is needed only once every 10 ms, you can see some data is collected and never used. We learned in previous modules that averaging can improve SNR. Consider this period measurement algorithm that averages all measurements in one 10-ms interval:

Initially, set **count** equal to zero. During an input capture interrupt

1. If **count** is 0, set **first** = time from TIMER_A3->CCTL[]
2. If **count** > 0, set **last** = time from TIMER_A3->CCTL[]
3. Increment **count**

During 10-ms periodic interrupt

1. If **count** < 2, set **period** = max value (too slow)
2. If **count** >= 2, set **period** = (**last**-**first**)/(**count**-1)
3. Set **count** equal to zero
4. Calculate **speed** from **period**

Lab: Tachometer

16.8 Which modules are next?

Module 17) Combine modules 12, 13, and 16 to create a control system that does spin the motors at a desired speed.

16.9 Things you should have learned

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

- Understand the relationship between duty cycle and speed, experiencing the effect of friction.
- Be able to use input capture to measure speed.
- Know how to use interrupts to build complex real-time systems.
- Know how to write and test a low-level software driver.

Module 17

Introduction: Control Systems

Introduction: Control Systems

Educational Objectives:

UNDERSTAND Basic concepts of a control system
INTERFACE The tachometer and a DC motor
CREATE An integral control system using feedback
DESIGN A differential drive robot that will move in a straight line

Prerequisites (Modules 10, 12, 13, 15)
- Periodic interrupts using SysTick (Module 10)
- Mechanical and electrical interfaces of motors (Module 12)
- Timer_A PWM output (Module 13)
- Timer_A input capture period measurement (Module 15)

Recommended reading materials for students:
- Volume 1 Sections 4.1, 9.4, and 9.7
  Embedded Systems: Introduction to the MSP432 Microcontroller, ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or
- Volume 2 Chapter 6
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

A **control system** is a collection of mechanical and electrical devices connected for the purpose of commanding, directing, or regulating a physical plant. The **state variables** are the properties of the physical plant that are to be controlled. In this module, we wish to spin the two motors at a prescribed speed. Thus, the state variable in this case will be motor speed. The **sensor** and **state estimator** comprise a data acquisition system. The goal of this data acquisition system is to estimate the state variables. We will attach tachometers to the motors so the system can measure speed of both motors. The estimated state variables, *X'(t)*, in this system will be the two measured speeds. The **actuator** is a transducer that converts the control system commands, *U(t)*, into driving forces, *V(t)*, that are applied the physical plant. We define the actuator command, *U(t)*, as the duty cycles for the PWM outputs to the two motors.

In general, the goal of the control system is to drive the real state variables to be equal to the desired state variables. In actuality though, the controller attempts to drive the estimated state variables to be equal the desired state variables. It is important to have an accurate state estimator, because any differences between the estimated state variables and the real state variables will translate directly into controller errors. We define the **error** as the difference between the desired and estimated state variables:

$$e(t) = X^*(t) - X'(t)$$

A closed-loop control system uses the output of the state estimator in a feedback loop to drive the errors to zero. The control system compares *X'(t)*, to the desired state variables, *X*(t)*, in order to decide appropriate action, *U(t)*. See Figure 1.

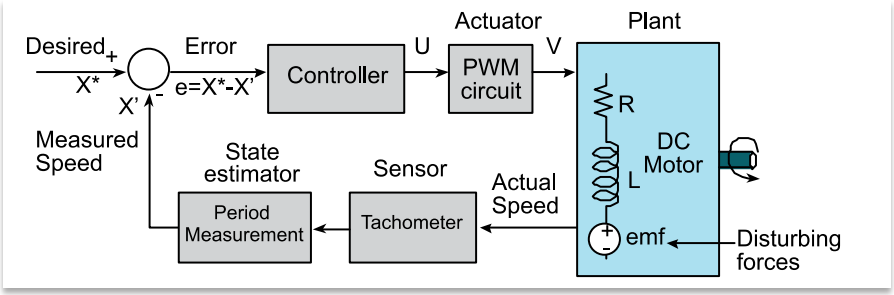


Figure 1. Block diagram of a MSP432-based closed-loop control system.

We can combine the period measurement from Module 15, the PWM output of Module 13, and the DC motor interface of Module 12 to build a motor controller. One effective yet simple control algorithm is an integral controller. We specify the actuator output as the integral of the accumulated errors.

$$U(t) = \int_0^t K_i E(\tau) d\tau$$

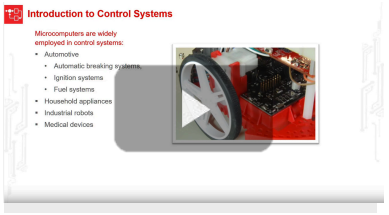
where K_i is a controller constant. For this controller, if the error is zero the actuator command remains constant. If the motor is spinning too slowly, the controller will increase power. If the motor is spinning too quickly, it will decrease power. For an integral controller, the amount of increase or decrease is linearly related to the error. So if the error is large it adds (or subtracts) a lot, and if the error is small it adds (or subtracts) a little.

17. TI-RSLK Module 17 – Control systems

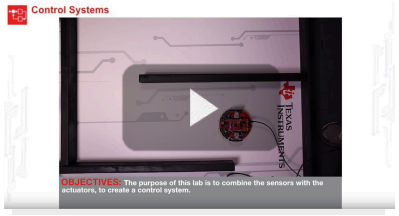
The purpose of this module is to create a control system by combining the sensors with the actuators. Incremental and integral control are simple algorithms for controlling motor speed.

Optionally, download all the curriculum documents for Module 17.

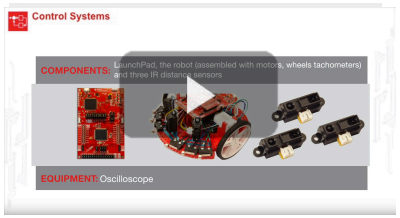
17.1 TI-RSLK Module 17 - Lecture video - Control systems
In this module you will learn the basic concepts of a control system.



17.2 TI-RSLK Module 17 - Lab video 17.1 - Demonstrating control system - integral control
You will learn an introduction to control.



17.3 TI-RSLK Module 17 - Lab video 17.2 - Demonstrating control system
Inputs, control equations and outputs.



Module 17

Lab 17: Control Systems

Lab: Control Systems

17.0 Objectives

The purpose of this lab is to develop a control system. In this module,

1. You will combine input capture measurements from Timer A3 and PWM outputs with Timer A0.
2. You will develop a system to control the speed of the two motors.
3. You will evaluate the performance of the control system.

Good to Know: Control systems are a rich and complex field within engineering spanning: electrical engineering, aerospace engineering, mechanical engineering, and computer engineering. This module provides a brief introduction.

17.1 Getting Started

17.1.1 Software Starter Projects

In addition to your solutions to Labs 13 and 15, look at this project:
Lab17_Control (starter project for this lab)

Note: Similar to Lab 14, you will find noise is a major problem for control systems. Continue to monitor the stability and accuracy of the tachometer measurements during this lab. Jittery measurements will cause even the most robust control system to fail.
The second issue with control systems is delay. Consider the closed loop between motor power -> motor speed -> tachometer measurement -> controller execution -> new duty cycle output. Delays within this loop (e.g., low pass filtering, slow controller execution rate) can cause the system to be unstable. Unstable systems produce oscillations.

17.1.2 Student Resources (in datasheets directory-Links)

MSP432P4xx Technical Reference Manual, Timer_A (SLAU356)
MSP432P401R Datasheet, msp432p401m.pdf (SLAS826)
MotorDriverPowerDistribution.pdf Data sheet for power board
Pololu Romi Chassis User's Guide.pdf How to build the robot

17.1.3 Reading Materials

Volume 1 Sections 4.1, 9.4, and 9.7
"Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Chapter 6
"Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

17.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	Romi Chassis Kit - Red	Pololu	3502
1	Motor Driver and Power Distribution Board for Romi	Pololu	3543
1	Romi Encoder Pair Kit, 12 CPR	Pololu	3542
2	Rechargeable Battery, Pack of 4, Metal Hydride 1300 mAh, 1.2V, AA	Energizer	626831
4	1.375in 4-40 Nylon standoff	Keystone	4809
2	0.187in 4-40 metal nut	Keystone	4694
6	0.5in 4-40 Nylon machine screw	Pololu	1962

17.1.5 Lab equipment needed

Oscilloscope (one or two channels at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling

Warning: Disconnect the VREG↔+5V wire when the LaunchPad USB cable is connected to the PC. Connect the VREG↔+5V wire when the robot is running on battery power. This way the motors always get power from the batteries, and not from the USB.

17.2 System Design Requirements

The goal of this lab is implement a control system to independently set the speed of the two motors. Let X^* be the desired speed (the units of X^* should match the units of the speed measurements obtained in Lab 16). Let $X'(t)$ be the estimated speed as implemented in Lab 16. We define the **controller error**, $e(t)$, to be the difference between the desired and estimated speed:

$$e(t) = X^* - X'(t)$$

The minimum desired speed should be larger than minimum speed measurable with your input capture system. The maximum desired speed should be the speed on the ground when the robot is moving with a duty cycle of 90%. The controller should be **stable,** meaning the robot moves with approximately constant speed. An **unstable** controller exhibits widely varying speeds oscillating between fast and slow.

The **accuracy** of the controller will be limited by the accuracy of the tachometer measurements. You will be required to measure accuracy, which we define as the average steady state error, but there is no requirement for this lab that the accuracy be less than a specific value.

The **stability** of the controller will be determined by the stability of the tachometer measurements and by the parameters of the controller. You will be required to measure stability, which we define as the standard deviation of the error, but there is no requirement for this lab that the stability be less than a specific value.

The **time constant** of the controller is defined as the time it takes to reach $(1-e^{-1})$ = 0.63 of the final speed given a change in desired speed. For example, if the current and desired speeds are 100 RPM and the desired speed is changed to 200 RPM, then the time constant is the time it takes to reach 163 RPM. You are required to measure time constant, but there is no requirement for this lab that the time constant be less than a specific value.

The ultimate goal of this lab is to be able to run the robot in a straight line at a desired and constant speed.

17.3 Experiment set-up

The construction of the robot has been performed in labs 5, 10, 12, 13, and 16. Refer back to these modules for more information on robot construction.

https://www.pololu.com/docs/0J68/all

The following table lists suggested pin connections for the tachometer and motor drivers.

LaunchPad (Ports)	MDPDB	Encoder	Description
P8.2/TA3CCP2	ELA	OUT A	Left Encoder A
P9.2/GPIO	ELB	OUT B	Left Encoder B
P10.4/TA3CCP0	ERA	OUT A	Right Encoder A
P10.5/GPIO	ERB	OUT B	Right Encoder B
P1.6	DIRR	PH	Right Motor Direction
P3.6	nSLPR	nSLEEP	Right Motor Sleep
P2.6	PWMR	EN	Right Motor PWM
P1.7	DIRL	PH	Left Motor Direction
P3.7	nSLPL	nSLEEP	Left Motor Sleep
P2.7	PWML	EN	Left Motor PWM

17.4 System Development Plan

17.4.1 Selection of the controller period

You will run the controller at a fixed rate using a periodic interrupt. Similar to sampling, running the controller at a regular rate allows you to implement digital signal processing. Let *Δt* be the period of the interrupt. For example, the integral equation

$$u(T) = \int_0^T a * e(t)dt$$

can be approximated as

$$u(T) = \sum_{n=1}^{T/\Delta t} a * e(n\Delta t)\Delta t$$

and implemented more simply as

$$u = u + a * e * \Delta t$$

There are multiple factors to consider when choosing a controller rate:

- The rate does not need be faster than the rate at which new speed data are obtained.
- Running the controller faster than the input rate is a waste of processor time because the controller equations will be executed multiple times with the same input data.
- The controller rate must be faster than the response rate of the motors. One rule of thumb is to choose the time interval for running the digital controller about 10 times slower than the time constant of the motor.
- Running the controller slower than the response time of the motor leads to instability.
- Running the controller faster will consume more processor time; running the controller slower allows for low pass filtering of the input data.

Note: Write your control software so it is easy to adjust the controller rate. This way you can experimentally test which rates work well for your robot.

17.4.2 Integral Controller

Write the two integral controllers that will run periodically within an Interrupt Service Routine (ISR). Use global variables to pass data into the controller. The two inputs to the left motor controller are **XstarL** (the desired speed) and **XprimeL** (the estimated speed). The output of the controller is the PWM duty cycle **UL** (e.g., 2 to 14998). For the left motor perform steps 1 – 5:

1. Read desired left motor speed: **XstarL**
2. Collect estimated left motor speed: **XprimeL**
3. Calculate error: **ErrorL** = **XstarL**- **XprimeL**
4. Calculate integral: **UL** = **UL** + (**A***__ErrorL__)/1024
5. Antireset windup: make sure 2 ≤ **UL** ≤ 14998

where **A** is a constant that defines the behavior of the integral controller. Perform similar steps for the right motor. Use signed 32-bit integer math.

After running the controller for each motor send outputs to the motor driver

```
Motor_Forward(UL,UR);
```

Compare the theoretical integral to the software implementation. The theoretical to software

$$u = u + a * e * \Delta t \leftrightarrow \textbf{UL} = \textbf{UL} + (\textbf{A}*\textbf{ErrorL})/1024$$

From this comparison you can see the software constant **A** is equivalent to *a*Δt*/1024.

Note: Consider the complete loop (motor power -> motor speed -> tachometer measurement -> controller execution -> new duty cycle output). Some delays are unavoidable, like the response time of the motor.

17.4.3 Tune the controller

Perform your initial tuning with the robot on blocks so the wheels do not touch the ground. For the initial value of **A**, take a large error value of 100 RPM and match it to a large change in duty cycle 10% (1500/15000). For example

A = 1024*1500/100 = 15,360

Start with a desired speed that you estimate to require a duty cycle of 50%. The first tuning will be for stability. Run the controller, and if the speed eventually stabilizes to more or less a constant then define it as stable. We define stability as the standard deviation of the error once it has reached steady state. It is unstable if

- The motor stops (0% duty cycle)
- The motor runs full speed (100% duty cycle)
- The motor oscillates fast and slow.

Saturated responses (stopped or full) are probably a result of a software bug or the sign that **A** is incorrect. Oscillations are probably a result of the **A** being two large. When initially searching for the best value of **A**, double or half the values of **A**, so you can quickly cover a wide range of values.

Once you have found a range of values that are stable, next you will tune for accuracy (average steady state error) and time constant (how quickly it stabilizes). Again, run this motor test on blocks so the wheels do not touch the ground. We define the **time constant**, τ, of the motor as the time it takes to achieve $(1\text{-}e^{-1}) = 0.63$ of the final speed, given a step change in desired speed to the motor. Read a switch on the LaunchPad and use this operator input to change the desired speed from typical (requiring about 50% duty cycle) to fast (requiring about 75% duty cycle). Use the debugger to observe error while running. If you performed Lab 11, you could plot speed versus time on the LCD.

Experiment to find the minimum and maximum speeds at which the controller is still stable and accurate. For this test run the robot on the ground. The goal is to run as straight as possible at more or less a constant speed.

17.4.4 Performance Evaluation

Write a test program that periodically collects motor speeds each time the controllers are run. Include the bumper driver from Lab 10 or Lab 14 so the robot stops on a collision. Dump desired speed, power (duty cycle) and speed data into buffers similar to Lab 10. For very long tests, you can dump into flash ROM. For shorter tests, you can dump into RAM. In the main program, perform these steps running the robot for 10 seconds.

1. Run forward at medium speed for 3 seconds
2. Run forward at fast speed for 4 seconds
3. Run forward at medium speed for 3 seconds
4. Stop the motors and stop the recording

Run this motor test on blocks and on a flat surface. We define the **time constant**, τ, of the motor as the time it takes to achieve $(1\text{-}e^{-1}) = 0.63$ of the final speed, given a step change in desired speed. Fit the speed versus time data to an exponential to estimate the time-constant of your controller.

$$y(t) = S_0 + \Delta S\, e^{-t/\tau}$$

where S_0, ΔS, and τ are least squares fit of the y(t) speed data verses time. Initial time is defined at the point the desired speed was changed.

17.5 Troubleshooting

Controller will not stabilize:

- Check the sign of **A** (too slow means increase duty cycle)
- Check the stability of the speed measurements given a constant duty cycle. If the inputs are noisy, the controller cannot function.
- Try an incremental controller. Let **K**=10 be a constant. Add **K** if too slow, subtract **K** if too fast.
- Check for overflow at multiply (**A***__ErrorL__). If this exceeds $\pm 2^{31}$, then reduce A and 1024.
- Check for underflow at divide by 1024. You will get zero if **A***__ErrorL__<1024. To solve, increase A.

Motors are very different:

- A little difference (±25%) is normal. A big difference may be friction or a bad motor.

17.6 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. The goal of this module is for you to understand Timer_A and its use for measuring period.

- In this lab we tuned the controller empirically. Why did we not use a mathematical model for the motor, and solve the optimal control parameters theoretically?
- There are three performance measures (accuracy, stability, and time constant) and only two adjustable tuning parameters (controller rate and A). From an engineering perspective what are the consequences of having so few parameters? Think about advantages and disadvantages of having only two parameters.
- What happens to your controller if the motor spins too slowly, e.g., less than 30 RPM?
- What happens to your controller if the motor stops, e.g., does not spin at all?
- How do we debug this system if the robot is moving along the ground?
- Why do performance measures (accuracy, stability, and time constant) differ if the robot is on blocks versus on the ground?

17.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. For example,

- If you completed Lab 11, add LCD outputs for each of the test functions. Remember to perform LCD output only in the main program and not during an ISR.
- If your robot does not have a tachometer to measure speed you could still perform this lab. For example if you have the IR distance sensors, then you could specify the desire to roll down the middle of the road. Assume the left and right IR sensors are measuring distance to the left and right walls, see Figure 1. For this controller we define error as the difference between distance to left and right. **Error = D_l-D_r**. Set the duty cycle of one wheel to a constant, and have the output of the controller determine the duty cycle of the other wheel.
- If your robot does not have a tachometer to measure speed you could still perform this lab. For example if you have the line sensor, then you could specify the desire to follow the line. Recall the output parameter for the **reflectance.c** driver in labs 6 and 10 was a number, where 0 meant on the line, positive numbers mean off center in one direction and negative numbers mean off center in the other direction. For this controller we define error simply as this reflectance measurement. Set the duty cycle of one wheel to a constant, and have the output of the controller determine the duty cycle of the other wheel.
- To improve time constant without affecting accuracy or stability, you could add a proportional term, implementing a PI controller.

1. Read desired left motor speed: **XstarL**
2. Collect estimated left motor speed: **XprimeL**
3. Calculate error: **ErrorL** = **XstarL**- **XprimeL**
4. Calculate integral: **UIL** = **UIL** + (**A***__ErrorL__)/1024
5. Antireset windup: make sure 2 ≤ **UIL** ≤ 14998
6. Calculate proportional: **UPL** = (**B***__ErrorL__)/1024
7. Combine: **UL** = **UIL**+**UPL**
8. Constrain: make sure 2 ≤ **UL** ≤ 14998

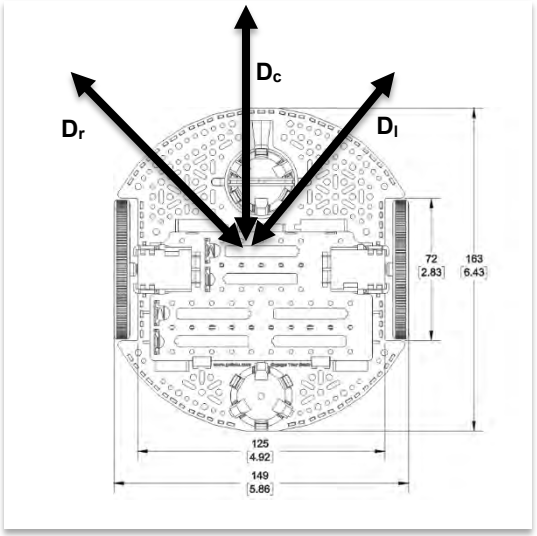


Figure 1. Define distance measured from a central point on the robot.

17.8 Which modules are next?

After this module, you are ready to solve any of the robot design challenges. If you wish to extend your robot to include wireless communication you have two options:
Modules 18 and 19) Add Bluetooth functionality.
Modules 18 and 20) Add Wifi functionality.

17.9 Things you should have learned

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

- Understand how the controller allows you to manage the uncertainties of friction.
- Know how to tune a digital controller empirically.
- Know how to use interrupts to build complex real-time systems. From a systems standpoint, your robot now has many components: bumper switches, line sensor, LCD, IR distance sensor, tachometer, and digital controller (PWM). You used a single main program for the non-real-time tasks like the LCD and operator buttons, but used interrupts for the real-time tasks.

Module 18

Introduction: Serial Communication

Introduction: Serial Communication

Educational Objectives:

UNDERSTAND The operation and use of first in first out queue
INTERFACE The robot to the PC using a serial channel
CREATE Two first in first out queues
DESIGN A command interpreter to assist in the robot challenge

Prerequisites (Module 10)
- Interrupts using SysTick (Module 10)

Recommended reading materials for students:
- Volume 1 Sections 4.5, 8.2, 11.3, and 11.4
  Embedded Systems: Introduction to the MSP432 Microcontroller, ISBN: 978-1512185676, Jonathan Valvano, copyright (c) 2017

or
- Volume Sections 2 3.4, 3.7, 4.9, and 5.6
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017

In this module you will develop an interrupting device driver using the Universal Asynchronous Receiver/Transmitter (UART). This serial port allows the microcontroller to communicate with devices such as other computers, input sensors, and output displays. Serial transmission involves sending one bit at a time, such that the data is spread out over time. The total number of bits transmitted per second is called the **baud rate**. Figure 1 shows the waveform produced when the MSP432 sends one byte of data.

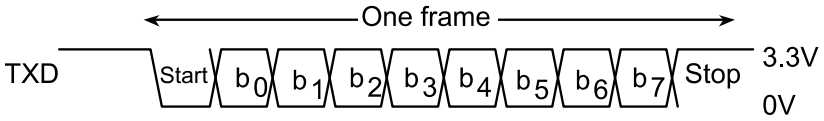


Figure 1. A serial data frame with 1 start bit, 8-bit data, 1 stop bit, and no parity bit.

The first in first out circular queue (FIFO) is quite useful for implementing a buffered I/O interface. It can be used for both buffered input and buffered output. The order preserving data structure temporarily saves data created by the source (producer) before it is processed by the sink (consumer). The class of FIFOs developed in this module will be statically allocated global structures. Because they are global variables, it means they will exist permanently and can be carefully shared by more than one thread. The advantage of using a FIFO structure for a data flow problem is that we can decouple the producer and consumer threads. Without the FIFO we would have to produce 1 piece of data, then process it, produce another piece of data, then process it. With the FIFO, the producer thread can continue to produce data without having to wait for the consumer to finish processing the previous data. This decoupling can significantly improve system performance.

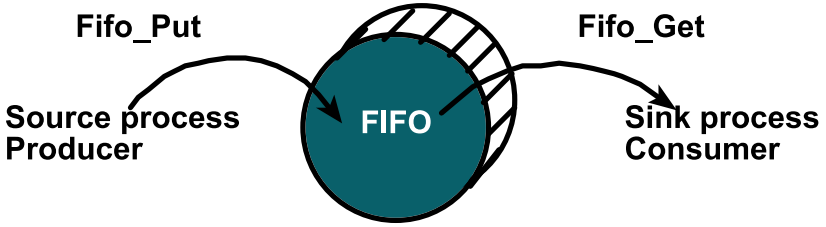


Figure 2. The FIFO is used to buffer data between the producer and consumer.

Because multiple threads are accessing the shared global structure, it is important to consider critical sections. If the put and get functions both read-modify-write the same global, an error would occur if one function started (read), was interrupted by the second, the second executes, and then the first completes (modify-write). For example, this approach has a critical section with the shared access to **Size**.

```
     Put              Get
Size=Size+1;  Size=Size-1;
```

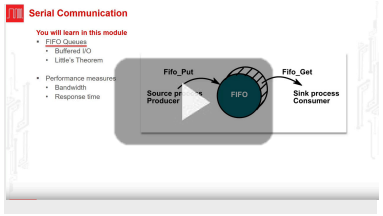
In summary, the UART driver will use interrupt synchronization with FIFO queues. This buffered approach will decouple the production of data from the consumption of data. For example, the main program can generate data to print, and put it into the FIFO queue. When the transmit UART hardware is idle, the ISR can get from the FIFO and write the data to the hardware. Buffering data in this matter is an efficient and effective mechanism for complex systems.

18. TI-RSLK Module 18 – Serial communication

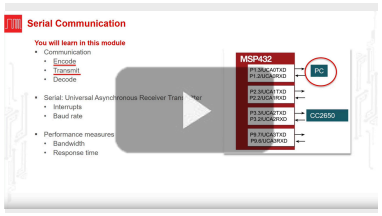
The purpose of this module is to understand the operation and use of first in first out (FIFO) queue to interface the robot to the PC using a serial channel. You will create two FIFO queues and design a command interpreter to assist in the robot challenge. You will develop an interrupting device driver using the universal asynchronous receiver/transmitter (UART). This serial port allows the microcontroller to communicate with devices such as other computers, input sensors, and output displays.

Optionally, download all the curriculum documents for Module 18.

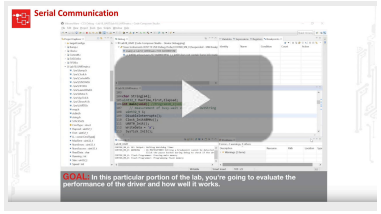
18.1 TI-RSLK Module 18 – Lecture video part I – Serial communication - FIFO
Learn FIFO queues, buffered I/O and Little's Theorem. Perform measures of bandwidth and response time.



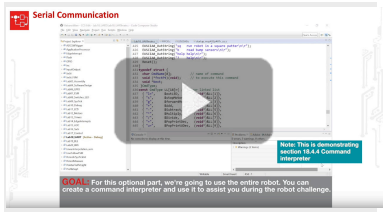
18.2 TI-RSLK Module 18 - Lecture video part II - Serial communication - UART
You will develop an interrupting device driver using the universal asynchronous receiver/transmitter (UART).



18.3 TI-RSLK Module 18 - Lab video 18.1 - Demonstrating UART
The purpose of this lab is to create an interrupt-driven UART driver.



18.4 TI-RSLK Module 18 - Lab Video 18.2 - Command interpreter
A command interpreter allows you to test multiple parts of your complex system.



Module 18

Lab 18: Serial Communication

Lab: Serial Communication

18.0 Objectives

The purpose of this lab is to develop an interrupt-driven software driver for the UART on the MSP432. In this module,

1. You will develop first in first out (FIFO) queues to stream data between foreground and background.
2. You will evaluate the performance of an interrupting UART driver.
3. You will design, develop, and test a command interpreter that can be used for the robot system.

Good to Know: Complex systems have a lot interweaved components. Streaming data from one module to another requires synchronization. FIFO queues are an effective mechanism to stream data without need to tightly couple execution of the two modules.

18.1 Getting Started

18.1.1 Software Starter Projects

Look at these two projects:
UART (busy-wait solution of the UART interface)
Lab18_UART (starter project for this lab)

18.1.2 Student Resources (in datasheets directory-Links)

MSP432P4xx Technical Reference Manual, Timer_A (SLAU356)
MSP432P401R Datasheet, msp432p401m.pdf (SLAS826)

18.1.3 Reading Materials

Volume 1 Sections 4.5, 8.2, 11.3, and 11.4
"Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 3.4, 3.7, 4.9, and 5.6
"Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller"

18.1.4 Components needed for this lab

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

In addition to the LaunchPad, you will use any of the robot features you have available to design a command interpreter.

18.1.5 Lab equipment needed

None

18.2 System Design Requirements

The goal of this lab is develop an interrupt-driven UART driver and use it to implement a command interpreter for the robot.

Note: When using the UART as a debugging mechanism, the time to execute functions like **EUSCIA0_OutUDec** and **EUSCIA0_OutString** determine the intrusiveness of the debugging output. With an interrupt-driven UART driver, if the FIFO queue is large enough and if the output rate is low enough, the FIFO never fills. If the FIFO never fills, no data is lost and the time to execute the output functions will be very short.

More specifically, you will develop two FIFO queues needed for the UART serial port driver. The **TxFifo0** streams output data from the main program to the UART ISR, and the **RxFifo0** streams input data from the UART ISR to the main program. You will find the prototypes in the header file **FIFO0.h**. Each FIFO has a buffer in permanent memory. The **Init** function initializes the FIFO, making it empty. The **Put** function stores data into the FIFO, and the **Get** function removes data from the FIFO. The FIFO preserves order; in other words, the order of data removed from the FIFO matches the order in which data is put. A buffer with 64 entries can contain 0 to 63 data items. Not allowing 64 items simplifies the distinction between empty (no items) and full (63 items). If the FIFO is full at the beginning of **Put**, the function returns with a full error. If the FIFO is empty at the beginning of **Get**, the function returns with an empty error.

The second requirement is to write an interpreter. The input comes from the keyboard, when running a terminal emulator like TExaSdisplay or PuTTy. Feel free to create your own syntax and list of commands. For example

If you type	*then the robot does*
Stop	The robot stops
Go	The robot goes straight
Back	The robot backs up
Left	The robot turns left
Right	The robot turns right
Slow	Set duty cycle to 2500
Fast	Set duty cycle to 7500
Sensor	Read and display sensor values

18.3 Experiment set-up

The UART data is streamed along the USB debugging cable. Therefore, the USB cable must be connected from robot to PC during this lab.

18.4 System Development Plan

18.4.1 Develop and test the FIFO queue

Implement the four FIFO functions that will be used to stream transmit data from the foreground to the UART ISR: **TxFifo0_Init**, **TxFifo0_Put**, **TxFifo0_Get**, and **TxFifo0_Size**. These functions can be tested with **Program 18_1**. In this test, the main program calls **Put** and the ISR calls **Get**. The data should be streamed in sequence and the FIFO never fills.

```
char WriteData,ReadData;
uint32_t NumSuccess,NumErrors;
void TestFifo(void){char data;
  while(TxFifo0_Get(&data)==FIFOSUCCESS){
    if(ReadData==data){
      ReadData = (ReadData+1)&0x7F; // in sequence
      NumSuccess++;
    }else{
      ReadData = data; // restart
      NumErrors++;
    }
  }
}
uint32_t Size;
int Program18_1(void){  // NumErrors should be zero
  uint32_t i;
  Clock_Init48MHz();
  WriteData = ReadData = 0;
  NumSuccess = NumErrors = 0;
  TxFifo0_Init();
  TimerA1_Init(&TestFifo,43);  // 83us, = 12kHz
  EnableInterrupts();
  while(1){
    Size = Random(); // 0 to 31
    for(i=0;i<Size;i++){
      TxFifo0_Put(WriteData);
      WriteData = (WriteData+1)&0x7F; // in sequence
    }
    Clock_Delay1ms(10);
  }
}
```

Note: We recommend you do not maintain a counter containing the number of items in the FIFO. Incrementing in counter during Put and decrementing the counter during Get will create a critical section when the two functions are used in a multithreaded system.

18.4.2 Performance measurements of OutString

The objective this section is to compare the busy-wait with interrupt driver. In both systems, strings of random size will be transmitted. The time to execute OutString is measured with SysTick. Since both versions have the same 115200 bits/sec baud rate, the actual time to perform the output will be identical. However, you will see how much shorter the execution time for the interrupt-driven version of OutString is as compared to the busy-wait version.

Compile and run **Program18_2**. Record the **MaxTime**, which is in usec.

```
char String[64];
uint32_t MaxTime,First,Elapsed;
int Program18_2(void){     // busy-wait OutString
  uint32_t i;
  DisableInterrupts();
  Clock_Init48MHz();
  UART0_Init();
  WriteData = 'a';
  SysTick_Init();
  MaxTime = 0;
  while(1){
    Size = Random(); // 0 to 31
    for(i=0;i<Size;i++){
      String[i] = WriteData;
      WriteData++;
      if(WriteData == 'z') WriteData = 'a';
    }
    String[i] = 0; // null termination
    First = SysTick->VAL;
    UART0_OutString(String);
    Elapsed = ((First - SysTick->VAL)&0xFFFFFF)/48;
    if(Elapsed > MaxTime){
      MaxTime = Elapsed;
    }
    UART0_OutChar(CR);UART0_OutChar(LF);
    Clock_Delay1ms(100);
  }
}
```

In a similar manner, compile and run **Program18_3**. This is essentially the same system, except the interrupt-driven version of **OutString** is used. Again, record the **MaxTime**. Because the FIFO never fills, the call to the **OutString** executes very quickly. Notice in **FIFO0.c**, each call to **TxFifo0_Put**, will measure FIFO size and implement a histogram. This histogram is a probability mass function (PMF), which counts the number of times each FIFO size has occurred. In the debugger, observe the contents of this histogram. You can use this measurement to predict maximum number of elements in the FIFO.

Note: There is an entire mathematical discipline called **Queuing Theory**. Central to this theory is the collection and interpretation of FIFO queue size data.

18.4.3 Create the second FIFO

Once you have fully debugged your **TxFifo0**, copy/paste this code to implement the **RxFifo0**. **Program 18_4** can be used to test both serial input and output.

18.4.4 Develop and test the interpreter

Write the main program that implements the interpreter. Feel free to adjust number of commands and the exact syntax of your interpreter. The purpose of the interpreter is to assist in solving the robot challenge.

One way to implement a command interpreter is to create a table that maps command name to the command function. For example this structure holds a string and a function pointer.

```
typedef struct {
  char CmdName[8];        // name of command
  void (*fnctPt)(void);   // to execute this command
}Cmd_t;
const Cmd_t Table[8]={
{ "Stop",   &doStop},
{ "Go",     &doGo},
{ "Back",   &doBack},
{ "Left",   &doLeft},
{ "Right",  &doFast},
{ "Slow",   &doSlow},
{ "Fast",   &doFast},
{ "Sensor"  &goSensor}};
```

where **doStop**, **doGo**, … etc are void-void functions that actually perform the associated commands. The interpreter reads a string by calling **EUSCIA0_InString**, and then searches the table for a match. If a match is found it executes the corresponding function.

18.5 Troubleshooting

There is no serial output:

- Run the UART project. It outputs at 115200 bps.
- There are two COM ports associated with the MSP432, use the lower number.

Can't open a COM port to the MSP432:

- Check the device manager for the COM port number.
- Sometimes CCS opens the COM port, preventing TExaSdisplay or PuTTy from access. Close CCS, unplug MSP432, plug in MSP432, start TExaSdisplay or PuTTy, open the COM port, and then start CCS.

TxFifo or RxFifo occasionally lose data:

- Make sure the FIFO properly handles empty on Get and full on Put.
- Make sure **Put** and **Get** do not write to the same shared global. This will cause a critical section. It is ok for **Get** to write to a global that **Put** reads. It is ok for **Put** to write to a global that **Get** reads.

Program 18_4 does not work:

- Retest the FIFO queues.

18.6 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. The goal of this module is for you to understand FIFO queues and their use in streaming data between threads.

- Consider an output channel that uses **EUSCIA0_OutString**. What does it mean if the **TxFifo0** is usually empty?
- Consider an output channel that uses **EUSCIA0_OutString**. What does it mean if the **TxFifo0** is usually full?
- Consider an input channel that uses **EUSCIA0_InString**. What does it mean if the **RxFifo0** is usually empty?
- Consider an input channel that uses **EUSCIA0_InString**. What does it mean if the **RxFifo0** is usually full?
- Assume you are streaming data between threads using a FIFO queue. You measure FIFO size periodically and calculate average FIFO size. Let **N** be the average number of elements in the FIFO (in characters). Assume you knew λ, the average rate at which data are sent (in characters/sec). Use **Little's Law** to estimate the average response time, which is how long data spends in the queue waiting to be sent.

18.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. For example,

- Run Program 18_3 with and without the histogram in order to estimate the overhead required to maintain the histogram.
- Implement the TxFifo0 in a second way (e.g., pointer and index). Use Program 18_3 to estimate the relative speed of the two methods.
- Learn about Kahn Process Networks (KPN). These networks use queues, and have a rich theory as long as none of the queues become full.

18.8 Which modules are next?

After this module, you are ready to solve any of the robot design challenges. If you wish to extend your robot to include wireless communication you have two options:

Module 19) Add Bluetooth functionality.
Module 20) Add Wifi functionality.

18.9 Things you should have learned

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

- Understand how the FIFO queue allows you stream data between threads on a complex system.
- Know how a PMF can be used to describe the behavior of a queue.
- Know how to use FIFO queues such that the queues never become full.

Module 19

Introduction: Bluetooth Low Energy

Introduction: Bluetooth Low Energy

Educational Objectives:

UNDERSTAND Basic concepts of Bluetooth Low Energy
INTERFACE The CC2650 to the MSP432 using UART communication
CREATE A BLE service with multiple characteristics
DESIGN A robot system that can be controlled by a smart device using BLE

Prerequisites (Module 18)

- Interrupting UART interface (Module 18)

Recommended reading materials for students:

- Volume 3 Sections 9.3, 9.4, 9.5, and 9.6
  Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers, ISBN: 978-1466468863, Jonathan Valvano, copyright (c) 2017

Bluetooth is wireless medium and a data protocol that connects devices together over a short distance. Examples of Bluetooth connectivity include headset to phone, speaker to computer, and fitness device to phone/computer. Bluetooth is an important component of billions of products on the market today. Bluetooth operates from 1 to 100 meters, depending on the strength of the radio. Most Bluetooth devices operate up to a maximum of 10 meters. However, in order to improve battery life, many devices reduce the strength of the radio, and therefore save power by operating across distances shorter than 10 meters. If the computer or phone provides a bridge to the internet, a Bluetooth-connected device becomes part of the Internet of Things (IoT).

Bluetooth is classified as a **personal area network** (PAN) because it implements communication within the range of an individual person. Alternatively, devices within a Bluetooth network are usually owned or controlled by one person. When two devices on the network are connected, we often say the devices are **paired**.

At the highest level, we see Bluetooth devices implement profiles. A **profile** is a suite of functionalities that support a certain type of communication. For example, the Advanced Audio Distribution Profile (A2DP) can be used to stream data. The Health Device Profile (HDP) is a standard profile for medical devices. There are profiles for remote controls, images, printers, cordless telephones, health devices, hands free devices, and intercoms. The profile we will use in this chapter is the **generic attribute protocol** (GATT). Within the GATT there can be once or more services.

Within a **service** there may be one or more characteristics. A **characteristic** is user or application data that is transmitted from one device to another across the network. One of the attributes of a characteristic is whether it is readable, writeable, or both. We will use the notify indication to stream data from the embedded object to the smart phone. Characteristics have a **universally unique identifier** (UUID), which is a 128-bit (16-byte) number that is unique. BLE can use either 16-bit or 32-bit UUIDs. A specific UUID is used within the network to identify a specific characteristic. Often a characteristic has one or more descriptors. Descriptors may be information like its name and its units. We will also see **handles**, which are a mechanism to identify characteristics within the device. A handle is a pointer to an internal data structure within the GATT that contains all the information about that characteristic. Handles are not passed across the Bluetooth network; rather, handles are used by the host and controller to keep track of characteristics. UUIDs are passed across the network.

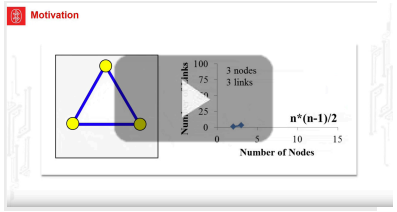
Simple Network Processor (SNP) is TI's name for the application that runs on the CC2650 when using the CC2650 with another microcontroller such as the MSP432. In this configuration the controller and host are implemented together on the CC2650, while the profiles and application are implemented on an external MCU. The application and profiles communicate with the CC2650 via the Application Programming Interface (API) that simplifies the management of the BLE network processor. The SNP API communicates with the BLE device using the Network Protocol Interface (NPI) over a serial (SPI or UART) connection. In this module, we will use a UART interface. This configuration is useful for applications that wish to add Bluetooth functionality to an existing device. In this paradigm, the application runs on the existing microcontroller, and BLE runs on the CC2650.

19. TI-RSLK Module 19 – Bluetooth low energy

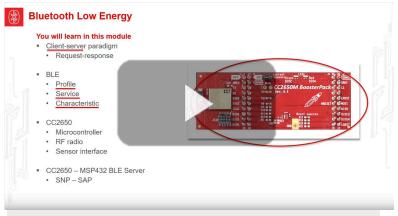
The purpose of this module is to understand basic concepts of Bluetooth® low energy (BLE). You will interface the TI SimpleLink™ BLE CC2650 Module BoosterPack™ Plug-in module to the SimpleLink MSP432P401R LaunchPad™ development kit using universal asynchronous receiver/transmitter (UART) communication. You will create a BLE service with multiple characteristics and design a robot system that can be controlled by a smart device using BLE.

Optionally, download all the curriculum documents for Module 19.

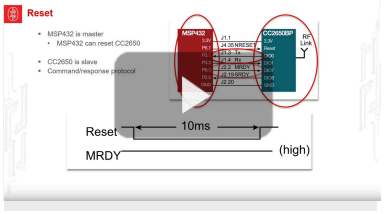
19.1 TI-RSLK Module 19 – Lecture video part I – Bluetooth Low Energy – Wireless
The purpose of this lab is to create an interrupt-driven UART.



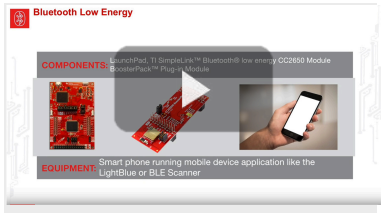
19.2 TI-RSLK Module 19 – Lecture video part II – Bluetooth Low Energy – Theory
The purpose of this module is to understand basic concepts of Bluetooth® low energy (BLE).



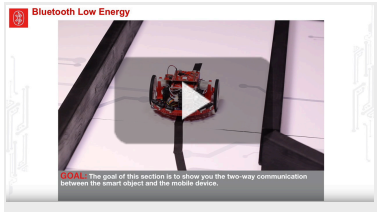
19.3 TI-RSLK Module 19 – Lecture video part III – Bluetooth Low Energy – SNP
Interface the TI SimpleLink™ BLE CC2650 Module BoosterPack™ Plug-in module to the SimpleLink MSP432P401R LaunchPad™ development kit UART.



19.4 TI-RSLK Module 19 - Lab video 19.1 - Demonstrating BLE
The purpose of this lab is to develop a robot system that can be controlled by a smart device.



19.5 TI-RSLK Module 19 - Lab video 19.2 - Communicating with the robot
Design a robot system that can be controlled by a smart device using BLE.



Module 19

Lab: Bluetooth Low Energy

Lab: Bluetooth Low Energy

19.0 Objectives

The purpose of this lab is to develop a robot system that can be controlled by a smart device. In this module,

1. You will send commands from the MSP432 to the CC2650 to establish a BLE link to a smart device.
2. You will use the BLE link to display sensor information from the robot to the smart device.
3. You will use the BLE link to send commands from the smart device to the robot.

Good to Know: Bluetooth Low Energy is a ubiquitous protocol used to wirelessly send and receive data between devices in the same room.

19.1 Getting Started

19.1.1 Software Starter Projects

Look at these three projects:
VerySimpleApplicationProcessor (a barebones BLE interface)
ApplicationProcessor (a BLE interface with abstraction)
Lab19_BLE (starter project for this lab)

Note: BLE is a complex protocol with a wide variety of features. In this module we have simplified BLE two ways. First, the low-level details of the radio and wireless communication are implemented on the CC2650 in a system called the Simple Network Processor (SNP). The high-level abstraction exists on the MSP432 as the Simple Application Processor (SAP). Second, this SAP-SNP system supports dozens of commands, but we will expose only the minimal set needed to establish a simple BLE link.

19.1.2 Student Resources (in datasheets directory-Links)

CC2650 Technical Reference Manual, (SWCU117)
CC2650 BLE Software Stack Developers Guide (SWRU393)
CC2650 Module BoosterPack (SWRU486)
CC2640_Simple_Network_Processer_API_Guide.pdf API Guide
SNP_API_Updated.pdf Shorthand guide to the NP-AP system

19.1.3 Reading Materials

Volume 3 Sections 9.3, 9.4, 9.5, and 9.6
Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers, ISBN: 978-1466468863, Jonathan Valvano, copyright (c) 2017

19.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	CC2650 BoosterPack	TI	BOOSTXL-CC2650MA

The CC2650 on the booster pack has been programmed with the simplified network processor (SNP) at the factory. You will need to have a smart device that can communicate via Bluetooth Low Energy.

19.1.5 Lab equipment needed

None

19.2 System Design Requirements

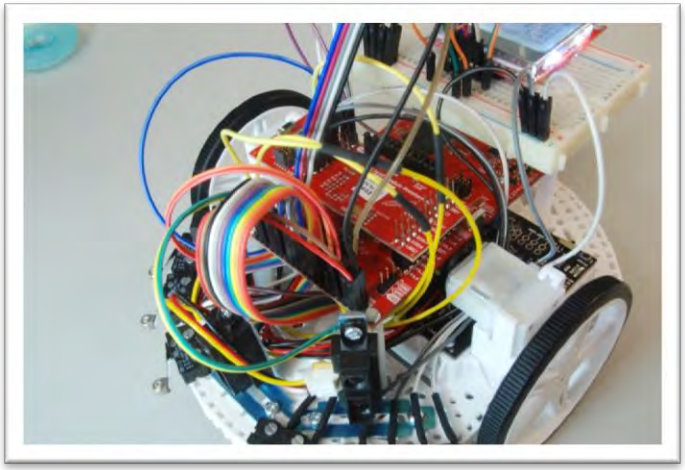
You will create a BLE link with at least two characteristics with read indications, which can be used to read sensor parameters of the robot.

Your BLE link will also have at least two characteristics with write indications, which can be used to write robot parameters like speed and commands.

You will create at least one characteristic with notify indication. Once activated on the smart device, you can stream data periodically or you can send data on an event like bump sensors recognizing a wall touch.

Lab: Bluetooth Low Energy

The ultimate goal of this lab is to be able to control the robot from the smart device using BLE.



19.3 Experiment set-up

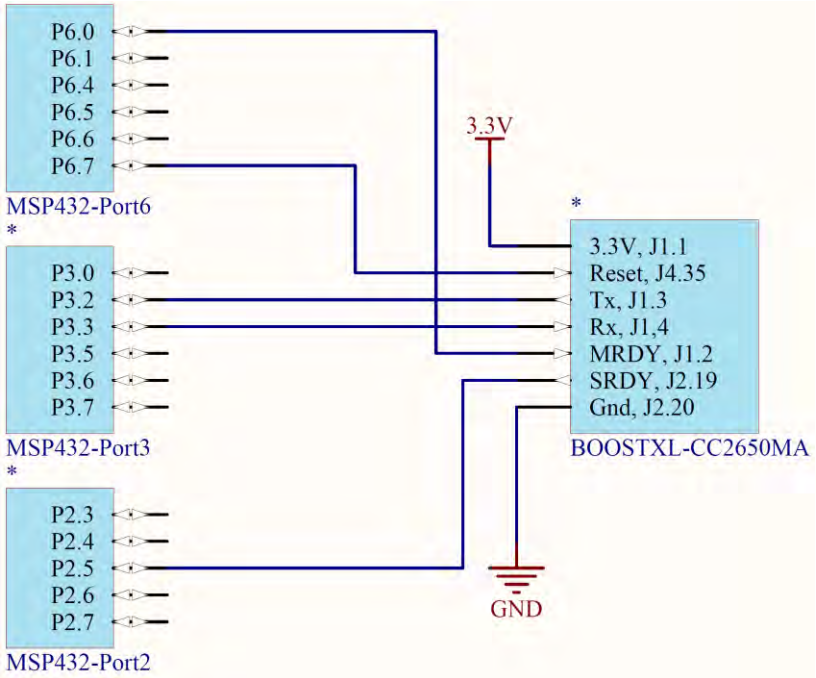
This lab will run with a wide range of BLE-enabled smart devices. For example:

iPhone running LightBlue
Android running BLE scanner.

You will need to attach the CC2650 BoosterPack to the MSP432 on the robot. The following table shows the pins used for the SNP-SAP system.

MSP432	SNP-SAP	CC2650	Description
P6.0 GPIO out	MRDY	DIO7	Master Ready
P2.5 GPIO in	SRDY	DIO8	Slave Ready
P6.7 GPIO out	NRESET	reset	Reset to CC2650
P3.3 UART TxD	RX	DIO1 RXS	MSP432 -> CC2650
P3.2 UART RxD	TX	DIO0 TXD	CC2650 -> MSP432

In addition to the above signals, 3.3V and ground from the MSP432 are used to power the CC2650 board. The details of the GPIO interface are described in the file **GPIO.c**. The details of the UART interface are described in the file **UART1.c**. The CC2650 also supports SPI interface, but this feature in not used in the lab, and the SPI pins are available for the robot.



19.4 System Development Plan

19.4.1 Run the VerySimpleApplicationProcessor project

For this section you need just the LaunchPad with the CC2650 BoosterPack attached. The first step in implementing your own BLE interface is to understand the SAP-SNP protocol. Attach the CC2650 to an MSP432 LaunchPad and build the **VerySimpleApplicationProcessor** project. Notice the 20 hard-coded message strings, which all start with **NPI_**. These are messages sent from the MSP432 to the CC2650 to configure BLE and perform communication. BLE goes through four phases. Notice these phases in the **main()** program.

Lab: Bluetooth Low Energy

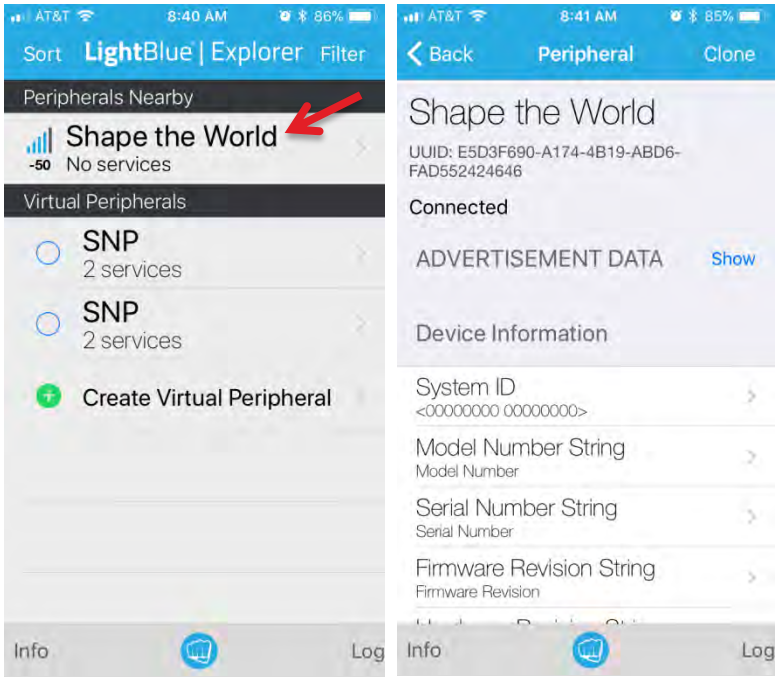
1) Hardware initialization. The call to **AP_Init** initializes the MSP432 interface pins (P3.2/P3.3 as UART, P6.0/P6.7 as GPIO output, and P2.5 as GPIO input), and issues a hardware reset to the CC2650. **AP_Init** will fail if the CC2650 is broken or missing.
2) Configure the CC2650 as a BLE server. Notice the commands to set the BLE device name, adds a service with four characteristics, registers the service, sets the parameters for advertisement and starts advertising.
3) Establishing the pairing. The CC2650/MSP432 smart object will be the slave. It advertises it is available for pairing. The smart device (cell phone) will be the master (client) and will initiate pairing. In this simple project, the main program runs the while loop until pairing has occurred.
4) Communication. Since the smart device is the master you will ask it to read and write indications for the four characteristics. This is a crude but simple way to read and write variables within the MSP432 from the smart device. The MSP432 **AP_RecvStatus** function returns a true when the BLE link sends an indication. The MSP432 **AP_RecvMessage** function returns that message describing the indication. The project has a simple and hard-coded way to process each possible indication.

Open a terminal program like TExaSdisplay in text mode. Build, debug, and run the **VerySimpleApplicationProcessor** project. Communication between the SNP (CC2650) and SAP (MSP432) is echoed to the PC on the UART0 channel (via USB cable). The first few lines of debugging output you should see on TExaSdisplay are

```
Very Simple Application Processor
 Reset CC2650
 Reset CC2650
 LP->SNP FE,03,00,55,04,1D,FC,01,B2
 SNP->LP FE,03,00,55,04,00,1D,FC,B3
 GATT Set DeviceName
 LP->SNP FE,12,00,35,8C,01,00,00,53,68,61,70,65,<…>,6C,64,DE
 SNP->LP FE,01,00,75,8C,00,F8
 NPI_GetStatus
 LP->SNP FE,00,00,55,06,53
 SNP->LP FE,04,00,55,06,00,00,00,00,57
 NPI_GetVersion
 LP->SNP FE,00,00,35,03,36
 SNP->LP FE,0D,00,75,03,00,01,10,00,02,02,00,00,91,<…>,00,EC
 Add service
 LP->SNP FE,03,00,35,81,01,F0,FF,B9
 SNP->LP FE,01,00,75,81,00,F5
 Add CharValue1
 LP->SNP FE,08,00,35,82,03,0A,00,00,00,02,F1,FF,BA
 SNP->LP FE,03,00,75,82,00,1E,00,EA
```

Note: The output **LP->SNP** shows a message from MSP432 to CC2650. The output **SNP->LP** shows a message from CC2650 to MSP432. Also notice that protocol typically involves a command/response behavior.

On the smart device (phone), open an application like **LightBlue**, and click the name of the MSP432/CC2650 BLE object, which has been programmed by the project **VerySimpleApplicationProcessor** to be called “Shape the World”. Once the client (phone) is paired with the server (MSP432/CC2650), you will see the “Connected” on the phone. On TExaSdisplay, you can see the messages sent between the MSP432 and CC2650 as this connection is established.

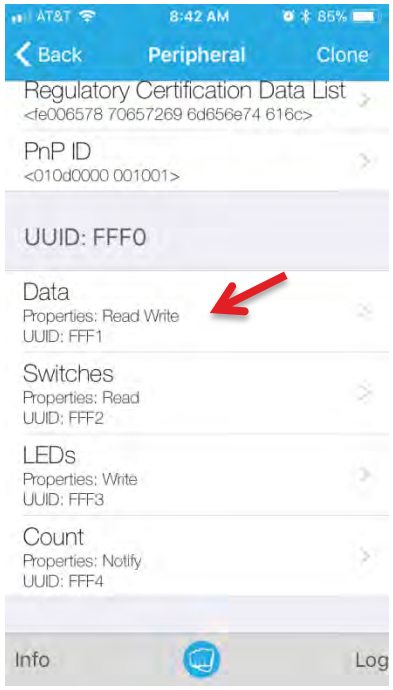


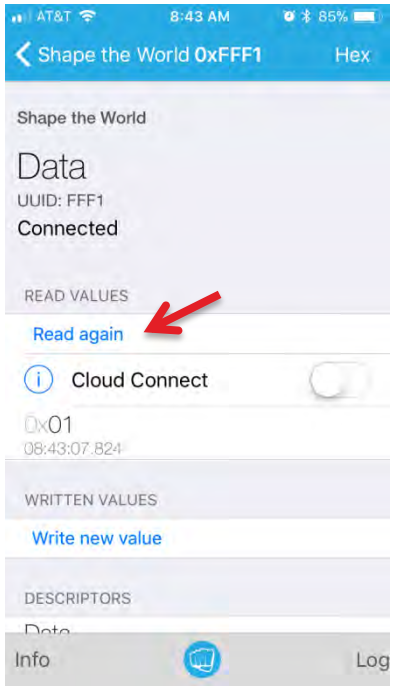
Next, scroll down and observe the four characteristics, which have been programmed by the project to be **Data, Switches, LEDs, and Count**. To interact with a characteristic, click on it. The Data characteristic has been programmed in this example for read and write properties, meaning information can flow both directions. Characteristics can be 1, 2, or more bytes. The Data characteristic

has been programmed in this example to be 1 byte. Once the characteristic window is open you can read the characteristic by clicking the “Read again”.

On **BLE Scanner** you see the characteristics listed by their UUID, which in this project will be 0000FFF1 0000FFF2 0000FFF3 and 0000FFF4. These four UUID numbers refer to **Data, Switches, LEDs,** and **Count** respectively.

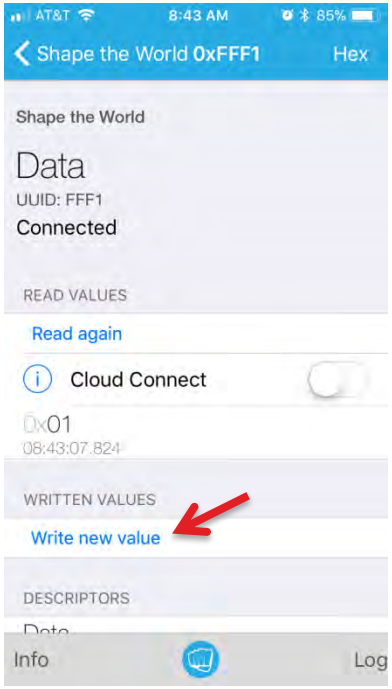
On TExaSdisplay, you can see the messages sent between the MSP432 and CC2650 as a read characteristic operation is performed.





You can write the characteristic by clicking “Write new value”. Writing a new value will open a dialog window, into which you type the new value. On **LightBlue**, the information is entered in hexadecimal. Once you have specified the value, click “Send” to write the information to the MSP432/CC2650 object.

On **BLE Scanner** you use the “byte array” format to write information from the smart device (phone) to the MSP432/CC2650 BLE object.



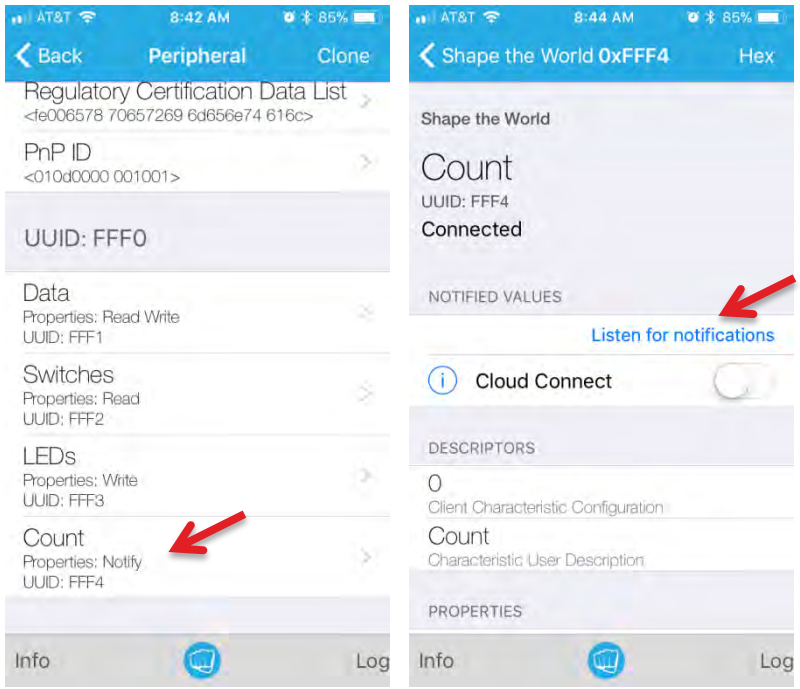
On TExaSdisplay, you can see the messages sent between the MSP432 and CC2650 as a write characteristic operation is performed.

Go back to the characteristic list and click the **Switches** characteristic. On the MSP432, press one of the LaunchPad switches and click “Read again”. You will be able to read the four possible values of the switches. (With BLE Scanner, the Switches characteristic has a UUID of 0000FFF2.)

Go back to the characteristic list and click the **LEDs** characteristic. On the smart device (phone) click “Write new value”. You will be able to send the eight possible values (0 to 7) to the LED. (With BLE Scanner, the LEDs characteristic has a UUID of 0000FFF3.)

We will create a characteristic with **notify** properties to stream information from the MSP432/CC2650 to the smart device (phone). Go back to the characteristic

list and click **Count**. (With BLE Scanner, the Count characteristic has a UUID of 0000FFF4.) On the smart device (phone) click “Listen for notifications”. This will configure the MSP432 to stream data to the smart object.



On TExaSdisplay, you can see the messages sent between the MSP432 and CC2650 as a notify characteristic operation is performed.

Notice in the main program loop that the BLE messages are handled.

19.4.2 Run the ApplicationProcessor project

Similar to the last section, you need just the LaunchPad with the CC2650 BoosterPack attached. In this example we will abstract the SAP-SNP protocol to create a more programmer-friendly software layer call an abstraction. Attach the CC2650 to an MSP432 LaunchPad and build the **ApplicationProcessor** project. This project runs in a similar

Notice the BLE interface is configured with a sequence of high-level function calls. See that each read/write characteristic has a global variable, a function to execute on read indication, a function to execute on a write indication. See that each notify characteristic has a global variable, a function to execute on a change of notification status (listen for notifications, stop listening).

```
r = AP_Init();
AP_GetStatus();  // optional
AP_GetVersion(); // optional
AP_AddService(0xFFF0);
AP_AddCharacteristic(0xFFF1,1,&ByteData,0x03,0x0A,
  "ByteData",&ReadByteData,&WriteByteData);
AP_AddCharacteristic(0xFFF2,2,&HalfWordData,
  0x01,0x02,"HalfWordData",&ReadHalfWordData,0);
AP_AddCharacteristic(0xFFF3,4,&WordData,
  0x02,0x08,"WordData",0,&WriteWordData);
  AP_AddNotifyCharacteristic(0xFFF4,2,&Switch1,
  "Button 1",&Button1);
AP_AddNotifyCharacteristic(0xFFF5,4,&Switch2,
  "Button 2",&Button2);
AP_RegisterService();
AP_StartAdvertisement();
```

Notice in the main program loop that the BLE messages are handled. The function **AP_BackgroundProcess();** must be called periodically to handle the read, write, and listen messages.

In a client-server paradigm, typically the client makes a request and the server answers. However, with a notify property, the server sends information to the client at times determined solely in the server. If the listen feature is active, the MSP432 calls **AP_SendNotification()** either periodically, as configured in this example, or it could be called at other times as your application needs.

Note: At the lowest layer of the SNP <-> SAP interface, the MSP432 interrupt synchronization to receive messages from the CC2650. Look **EUSCIA2_IRQHandler** in the UART1.c file. No BLE data is lost if the call to **RxFifo_Put** never results in a full FIFO. Refer back to module 18 for the importance of FIFOs in complex systems.

19.4.3 Low-level software development

There are a couple of low-level functions you need complete for this lab. In the file **AP.c** you need to create **NPI_SetAdvertisementDataJacki**, which will be a hard-coded message to specify the advertising name of your object. For an example, see **NPI_SetAdvertisementData**, which was used for the **ApplicationProcessor** project. For a detailed description of this message, see the 0x55,0x43 “Set Advertisement Data” command in the SNP API guide CC2640_Simple_Network_Processer_API_Guide.pdf.

Next, you need to implement the **AP_StartAdvertisementJacki** function in **AP.c** that uses the **NPI_SetAdvertisementDataJacki** message to start advertising. For an example, see the function **AP_StartAdvertisement**, which was used for the **ApplicationProcessor** project.

19.4.4 High-level software development

Make a list of the robot sensors you wish to communicate. Choose whichever sensors you plan to use during the robot challenge, and configure them as read-indication characteristics:

1. Bump sensors
2. Line sensor
3. IR distance sensors
4. Tachometer

Choose parameters you might which to set during the robot challenge, and configure them as write-indication characteristics:

1. Default duty-cycle to PWM
2. Controller setpoint and/or gain
3. Robot function commands (go, stop, turn, etc.)

Choose parameters you might which to stream during the robot challenge, and configure them as notify characteristics:

1. Controller error(s)
2. Controller intermediate decisions
3. Strategic sensor data

Combine software from previous systems to create a BLE-enabled robot system. Again, look ahead to the robot challenge and implement BLE features that will assist in debugging the challenge.

19.5 Troubleshooting

BLE will not communicate:

- The two projects **VerySimpleApplicationProcessor** and **ApplicationProcessor** should run without hardware or software modifications. The SNP<->SAP messages can be viewed on TExaSdisplay.
- The MSP432 needs to have these five pins free to implement communication with the CC2650 P6.0, P2.5, P6.7, P3.3, and P3.2. Make sure there is no other hardware connected to these pins.
- There is a way to reflash the CC2650 with the SNP software. See end of lab for details,

Data looks funny:

- Make sure the size of the characteristic (1 2 or 4 bytes) matches the size of the variable uint8_t uint16_t or uint32_t.
- Recall the LightBlue application read and writes in hexadecimal.

19.6 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. The goal of this module is for you to have a brief introduction to BLE.

- In this system which is the client and which is the server? How is a client different from a server?
- Why is this system called a personal area network?
- You should have a clear understanding between a profile, service, and characteristic.
- What are handles, and how are they used in this system?
- What is the advantage of interrupt driven receiver communication on this system? E.g., an incoming message from the CC2650 to the MSP432 causes interrupts on the MSP430.
- What are the advantages and disadvantages of implementing this system using two microcontrollers: MSP432 and CC2650? Compare this approach to implementing the entire robot on the CC2650 LaunchPad.

Lab: Bluetooth Low Energy

19.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. For example,

- This system is a personal area network. How can it be extended to be an Internet of Thinks object? Explore the Cloud Connect feature of the smart device (phone).
- This lab used an existing application (e.g., LightBlue) in the client. Explore the steps to creating a custom application.
- Search TI.com for information on **SensorTag**. This is a rich development environment (parts, boards, and software) for BLE systems involving the CC2640.

19.8 Which modules are next?

After this module, you are ready to solve any of the robot design challenges. If you wish to extend your robot to include wifi communication you complete: Module 20) Add Wifi functionality.

19.9 Things you should have learned

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

- Understand the basic concepts in BLE communication.
- Know profile, service, characteristic, client and server.
- Know how to use interrupts simplify software develop on complex systems.

19.10 Reflash the CC2650

This should not be needed. It should be used only as a last resort.Step 0) Create an account on https://my.ti.com/ and log in.

Step 1) Search TI.com for "**Smartrf flash programmer**". Download and unzip a file called **flash-programmer-2-1.7.5.zip**. In administrator mode, install the application, **Setup_SmartRF_Flash_Programmer_2.exe**

Step 2) Download and unzip hex files from this web link
ble_2_02_simple_np_setup.exe

http://software-dl.ti.com/dsps/forms/self_cert_export.html?prod_no=ble_2_02_simple_np_setup.exe&ref_url=http://software-dl.ti.com/lprf/BLE-Simple-Network-Processor-Hex-Files

These hex files (object code) implement the BLE stack in the form of the **simple network processor** (SNP). This download creates two directories: one with files for the BoosterPack (cc2650bp) and one with files for the LaunchPad (cc2650lp).

Step 3) Find this hex file on your computer:
simple_np_cc2650bp_uart_pm_xsbl.hex
Notice the letters **bp** (for BoosterPack) **uart** means serial communications, **pm** means hardware handshake and **xsbl** means no serial bootloader.

Step 4) Use the Flash Programmer to burn this hex file onto your CC2650 BoosterPack. The MSP432 LaunchPad can be the debugger/loader for the CC2650.

19.11 Using the CC2650 LaunchPad

Follow steps 0, 1, 2 from Section 19.10

Step 3) Find this hex file on your computer:
simple_np_cc2650lp_uart_pm_xsbl.hex
Notice the letters **lp** (for LaunchPad) **uart** means serial communications, **pm** means hardware handshake and **xsbl** means no serial bootloader.

Step 4) Use the Flash Programmer to burn this hex file onto your CC2650 BoosterPack. The CC2650 LaunchPad can be programmed by simply plugging in its USB, like other LaunchPad.

Module 20

Introduction: Wi-Fi

Introduction: Wi-Fi

Educational Objectives:

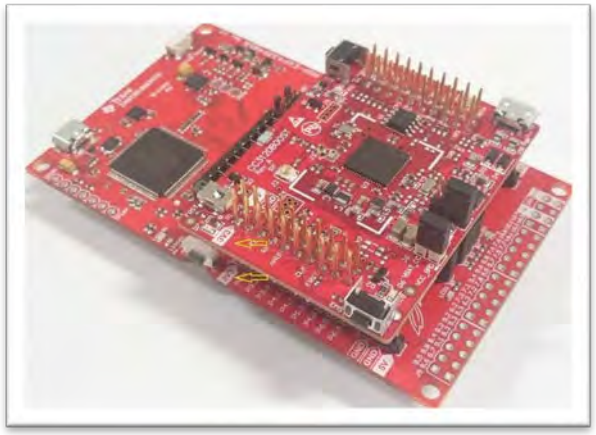
REVIEW Synchronous serial communication
UNDERSTAND basic RTOS concepts
DEVELOP a set of Wi-Fi communication functions
LEARN how to interact with web services
DESIGN, BUILD & TEST A SYSTEM
Interface a Wi-Fi radio module to the microcontroller

Prerequisites (Modules 1, 4, 6, 11, 14, and 18)

- Running code on the LaunchPad using CCS (Module 1)
- Basic C programming (Module 4)
- GPIO (Module 6)
- Interface LCD (Module 11)
- I/O Triggered Interrupts (Module 14)
- Serial Communications (Module 18)

Recommended reading materials for students:

- Volume 2 Sections 11.3, and 11.4
  Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller, ISBN: 978-1514676585, Jonathan Valvano, copyright (c) 2017
- Volume 3 Chapters 3, 4, and 5
  Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers, ISBN: 978-1466468863, Jonathan Valvano, copyright (c) 2017



Wi-Fi (short for “Wireless Fidelity”) is ubiquitous in modern embedded systems. With more devices requiring a direct connection to the internet, the Wi-Fi standard is a popular option and by many criteria the easiest option to create IoT applications. Wi-Fi radios make use of the SPI (see Module 11) or can in some instances be driven through AT commands with UART (see Module 18). The synchronous peripheral interface (SPI) system can operate as a master or as a slave. The channel can have one master and one slave, or it can have one master and multiple slaves. In this module, the MSP432 will be the master and the Wi-Fi module will be the slave. The master initiates all data communication.

Wi-Fi requires a network stack to manage the connections. A network or protocol stack is the software implementation of the communication protocols and is common for most types of RF communication. Sometimes this stack can be implemented on the main microcontroller or sometimes is can be running on the RF module, leaving more memory for the application code on the primary microcontroller. In this module, the MSP432 will make use of the SimpleLink SDK connectivity drivers to control the CC3120 Wi-Fi radio.

The CC3120 communicates with the MSP432 over SPI. The SPI protocol includes four I/O lines. The slave select STE is an optional negative logic control signal from master to slave signal signifying that the channel is active. The second line, CLK, is a 50% duty cycle clock generated by the master. The slave in master out (SIMO) is a data line driven by the master and received by the slave. The slave out master in (SOMI) is a data line driven by the slave and received by the master. In order to work properly, the transmitting device uses one edge of the clock to change its output, and the receiving device uses the other edge to accept the data.

In the lab associated with this module, we will interface a CC3120 using the SimpleLink SDK APIs (Application Programming Interface). APIs are specialized functions provided by software tools to interface or pass data. In this case, TI provides API access to the CC3120 Wi-Fi radio that we can use with the MSP432 very easily. We will also need to connect our system to the cloud. This can be done in a near infinite amount of ways by connecting to available web services or creating your own client and server implementations.

The SimpleLink SDK leverages a different software structure that is called a Real-Time Operating System or RTOS. The RTOS will help us manage the complexity of the application that now includes Wi-Fi communication. We will learn a few RTOS concepts with the goal of helping us implement the SimpleLink Wi-Fi.

Module 20

Lab: Wi-Fi

Lab: Wi-Fi

20.0 Objectives

The purpose of this lab is to interface a Wi-Fi radio to the microcontroller and connect to cloud services.

1. You will connect a Wi-Fi radio to the microcontroller.
2. You will set up TI-RTOS
3. You will use the synchronous serial protocol to communicate.
4. You will connect the system to cloud services.

Good to Know: There are many possible applications you can implement once your system is connected to the internet. You will be able to provide remote access to the robot or give it data from external sources or sensors that can potentially feed into its control logic.

20.1 Getting Started

20.1.1 Software Starter Projects

Look at these projects:
network_terminal_MSP_EXP432P401R_tirtos_ccs (from TI download)
Lab20_WiFi (starter project for this lab)

20.1.2 Student Resources (in datasheets directory-Links)

CC3120.pdf (SimpleLink™ Wi-Fi Wireless Network Processor)

20.1.3 Reading Materials

Volume 2 Sections 11.3 and 11.4
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller",
and
Volume 3 Chapters 3, 4 and 5
Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers

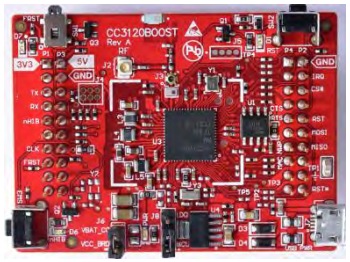


Figure 1. CC3120 Wi-Fi BoosterPack.

20.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R
1	CC3120 Wi-Fi BoosterPack	TI	CC3120BOOST

20.1.5 Lab equipment needed

Wi-Fi router (with internet connection) or cellular hotspot

20.2 System Design Requirements

The overall goal of this lab is to interface a Wi-Fi radio to the microcontroller and use it to connect to the local Wi-Fi router and then interact with a cloud service.

A Wi-Fi device can operate in two main modes. **Station mode** allows it connect to a local access point (AP). For example, your smart home device connects to your house Wi-Fi router or your cell phone connects to the airport Wi-Fi network. The device is in station mode and the router is in AP mode. **AP mode** lets the device act as an access point with a broadcast SSID that other devices can connect to. This is most commonly used by your local router but can also be used by a device if we need to do some first time setup or if we only need local WAN information and no internet access is required. An AP has a limited number of connections it can service at any given time. High end routers can handle hundreds of connections. The CC3120 can handle four simultaneous connections in AP mode.

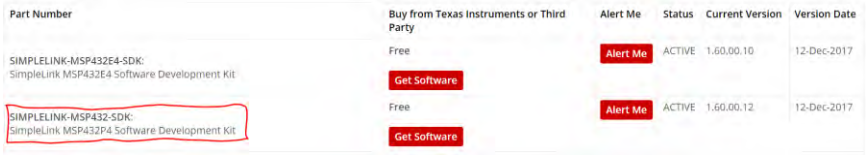
Using SimpleLink Wi-Fi is a more complex operation than what we have done in previous labs. To use SimpleLink Wi-Fi to its full ability, we will make use of a Real-Time Operating System (RTOS) specifically TI-RTOS. TI-RTOS is a free to use RTOS available from TI and optimized on TI processors. This module will not go too deeply into RTOS concepts, but we will be using TI-RTOS to enable our Wi-Fi communication and give you some exposure to how using an RTOS in a system is initialized.

20.3 Experiment set-up

In this first section we are going to setup the SimpleLink SDKs in our CCS environment. You will need to download and install two SDKs for use in your workspace.

TI provides a SimpleLink SDK (software development kit) to enable development with the MSP432 and provide many additional options for the software development of the microcontroller.

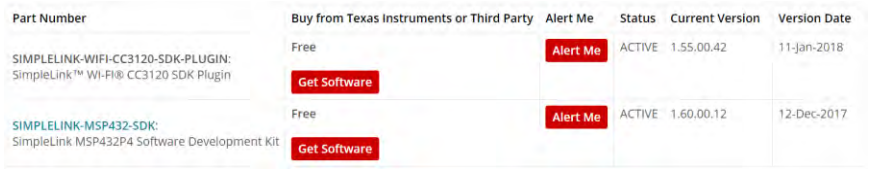
First, download the SimpleLink MSP432P4 SDK from http://www.ti.com/tool/SIMPLELINK-MSP432-SDK and download version 1.60.00.12 or later. Be sure to download the SDK for the MSP432P4 and not the E4. Run the downloaded application to install the plugin. Alternatively, you can also download it from the Resource Explorer front page (double check the version number). To access Resource Explorer go to the top menu View → Resource Explorer. You may need to restart CCS to complete the installation.



SimpleLink plugins are intended to extend functionality of each individual platform SDK to include specialized use-cases such as adding wireless functionality.

While all of the plugins have the same basic structure and look-and-feel of an SDK, they are not meant as standalone applications and rely heavily on components from the platform SDK. The SimpleLink Wi-Fi SDK Plugin, for example, relies heavily on the TI-Drivers and RTOS kernel components from the MSP432P4 SDK.

Second, download the SimpleLink Wi-Fi SDK Plugin from http://www.ti.com/tool/simplelink-wifi-cc3120-sdk-plugin and download version 1.55.00.42 or later. Be sure to download the SIMPLELINK-WIFI-CC3120-SDK-PLUGIN. Run the downloaded application to install the plugin. Alternatively, you can also download it from Resource Explorer front page (double check the version number). To access Resource Explorer go to the top menu View → Resource Explorer. You may need to restart CCS to complete the installation.



Third, run those downloaded executable files to start the installer. Use the default location C:\ti as the destination folder. When complete both SDKs should reside in the C:\ti\simplelink_msp432p4_sdk_1_60_00_12 and C:\ti\simplelink_sdk_wifi_plugin_1_55_00_42 file paths or similar. Restart your CCS session and those new SDKs should be detected by the IDE.

In this next section, we will do a TI-RTOS specific requirement of importing the kernel project to our workspace. The kernel is the main code base required to run the RTOS or any type of modern operating system. This project contains the TI-RTOS kernel and is needed for our Wi-Fi example. The kernel build project comes in a variety of flavors such as release and debug, tirtos and nortos, ccs and gcc compilers. For this setup you must choose the release version of the **tirtos kernel** using the **ccs compiler**. Start CCS and execute File → Import → CCS Projects and go into the file path for the MSP432P4 SDK **C:\ti\simplelink_msp432p4_sdk_1_60_00_12\kernel\tirtos\builds\MSP_EXP432P401R\release\ccs**
Click finish to import. With the TI-RTOS kernel properly imported we are ready to utilize the TI-RTOS based examples in our CCS workspace. We will make use of this in section 20.4.3.

Lab: Wi-Fi

The SimpleLink Wi-Fi SDK Plugin is designed for development on the CC3120 Network Processor and MSP432 Host MCU. The CC3120 and MSP432 communicate over the SPI or UART host interface. We will use SPI. The CC3120 requires an external host MCU for the user application.

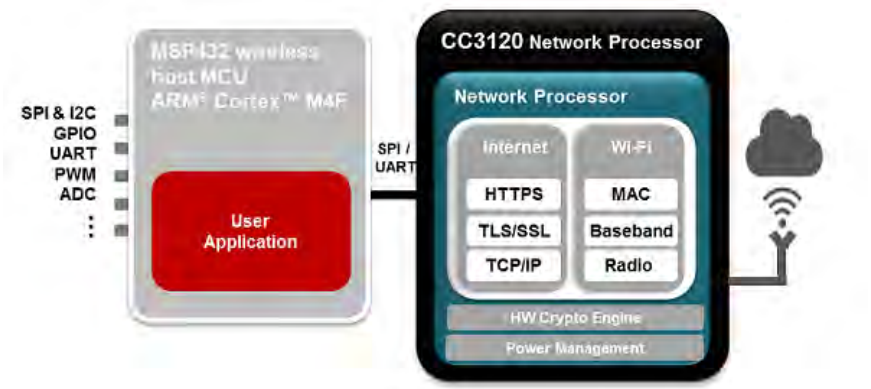


Figure 2. Block Diagram of MSP432 and CC3120 interface.

We will set up the hardware. Disconnect your MSP432 from the robot and Remove any BoosterPacks from previous modules.

1. Mount the CC3120 BoosterPack on top of a MSP432 LaunchPad so the pins align as shown. Be careful not to bend any pins.

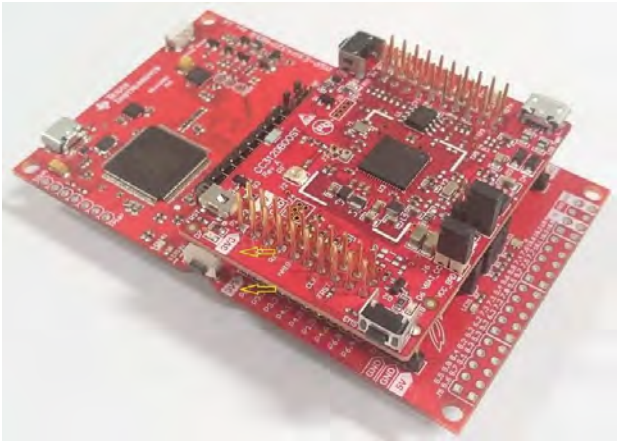


Figure 3. Wi-Fi BoosterPack positioned on top of MSP432 with markers aligned.

2. Plug a micro-USB connector to the MSP432 board. This is used as a power source and the programming interface for the user application. There is no need to plug in to the USB connector on the BoosterPack, this is used for external power source but not needed in the context of this application. You just want one USB cable to the MSP432 LaunchPad like the previous labs.

You will implement this lab using the MSP432 LaunchPad and the CC3120 Wi-Fi BoosterPack. Note that the TI documentation references the CC31XXEMUBOOST for updating firmware, but you will not need that to complete the activities in TI-RSLK.

Tables 4 through 7 show the pins used by the MSP432-CC3120 interface. When extending the system beyond the activities in this lab, you can use any of the pins marked unused for additional interfaces.

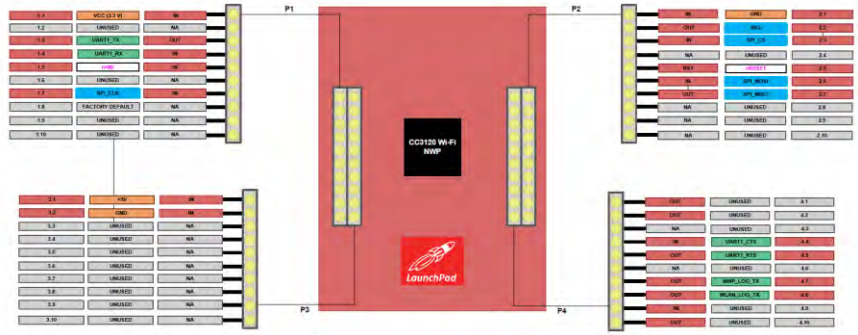


Figure 4. Pin Connections diagram

Pin Number	Signal Name	Direction	Pin Number	Signal Name	Direction
P1.1	VCC (3.3 V)	IN	P2.1	GND	IN
P1.2	UNUSED	NA	P2.2	IRQ	OUT
P1.3	UART1_TX	OUT	P2.3	SPI_CS	IN
P1.4	UART1_RX	IN	P2.4	UNUSED	NA
P1.5	nHIB	IN	P2.5	nRESET	IN
P1.6	UNUSED	NA	P2.6	SPI_MOSI	IN

Table 5. J1 Pin Connections

Pin Number	Signal Name	Direction	Pin Number	Signal Name	Direction
P1.7	SPI_CLK	IN	P2.7	SPI_MISO	OUT
P1.8	FACTORY DEFAULT	NA	P2.8	UNUSED	NA
P1.9	UNUSED	NA	P2.9	UNUSED	NA
P1.10	UNUSED	NA	P2.10	UNUSED	NA

Table 6. J2 Pin Connections

Pin Number	Signal Name	Direction	Pin Number	Signal Name	Direction
P3.1	+5 V	IN	P4.1	UNUSED	OUT
P3.2	GND	IN	P4.2	UNUSED	OUT
P3.3	UNUSED	NA	P4.3	UNUSED	NA
P3.4	UNUSED	NA	P4.4	UART1_CTS	IN
P3.5	UNUSED	NA	P4.5	UART1_RTS	OUT
P3.6	UNUSED	NA	P4.6	UNUSED	NA
P3.7	UNUSED	NA	P4.7	NWP_LOG_TX	OUT
P3.8	UNUSED	NA	P4.8	WLAN_LOG_TX	OUT
P3.9	UNUSED	NA	P4.9	UNUSED	IN
P3.10	UNUSED	NA	P4.10	UNUSED	OUT

Table 7. J3 and J4 Pin Connections

20.4 System Development Plan

20.4.1 Loading the Network Terminal Program

Primary: Import from SimpleLink CC3120 SDK Plugin

1. File → Import → CCS Project and navigate to C:\ti\simplelink_sdk_wifi_plugin_1_55_00_42\examples\rtos\MSP_EXP432P401R\demos\network_terminal
2. Click OK
3. Build the project by selecting *Build Project* from the Project menu or right-clicking the name of the project. It may take a couple minutes to build.
4. **Using CCS Debugger:** Start a Debug session by clicking the green bug in the top menu. Your MSP432 LaunchPad will need to be plugged in.
5. Open a UART terminal (or two) on your device's COM port. We want to use the XDS110 Class Application/User UART port with the following parameters:

   UART Configuration
   Baud rate: 115200
   Data: 8 bit
   Parity: None
   Stop: 1 bit
   Flow control: None

   To open in CCS go to View > Other… > Terminal > Terminal. Open a new Terminal with the above configuration

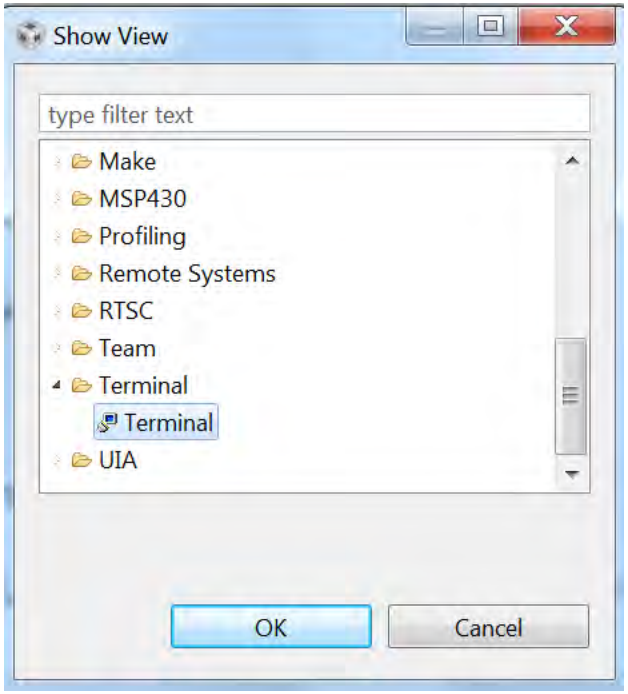


Figure 8. Show view options

Connection type: Serial Port
Click New connection. In the Launch Terminal pop up select Serial Terminal and the COM port. Confirm the configuration is the correct Baud and click ok.

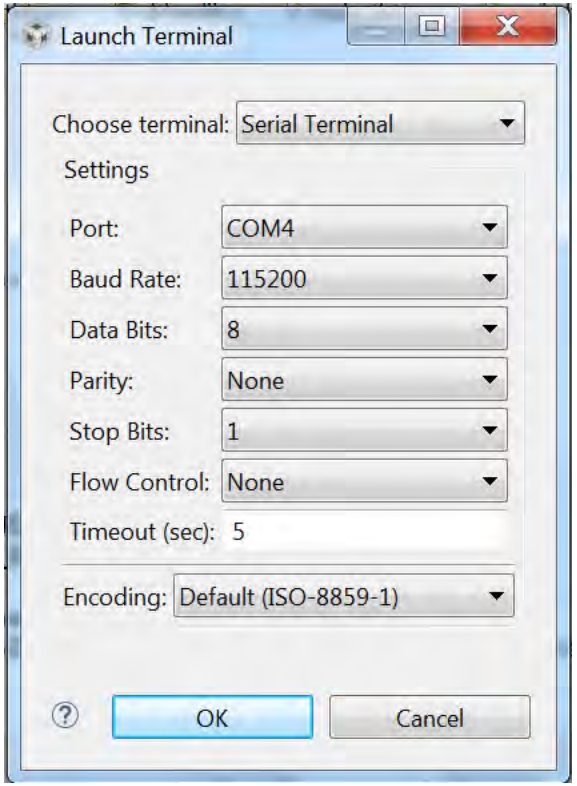


Figure 9. Serial Terminal options

6. Using the CCS debugger, click the green arrow in the top menu to start executing your code. You can now access the Wi-Fi commands by typing them in the terminal. We will explain in 20.4.2 the available commands and then use these to ping TI's website.

Lab: Wi-Fi

Alternative procedure to load example: Using Resource Explorer

1. In CCS, open the TI Resource Explorer (View → Resource Explorer)
2. Type in MSP432 in top search bar and select the MSP432P401R LaunchPad to filter the results to our LaunchPad board.
3. Find the SimpleLink SDK Plugins folder. Expand the connectivity folder and Wi-Fi plugin folder. Expand the folders as shown to select the network_terminal example (Examples → Development Tools → MSP432P401R LaunchPad → Demos → network_terminal → TI-RTOS → CCS Compiler → network_terminal), then click the Import to IDE icon at the top-right to download the code to the IDE and install any dependencies.

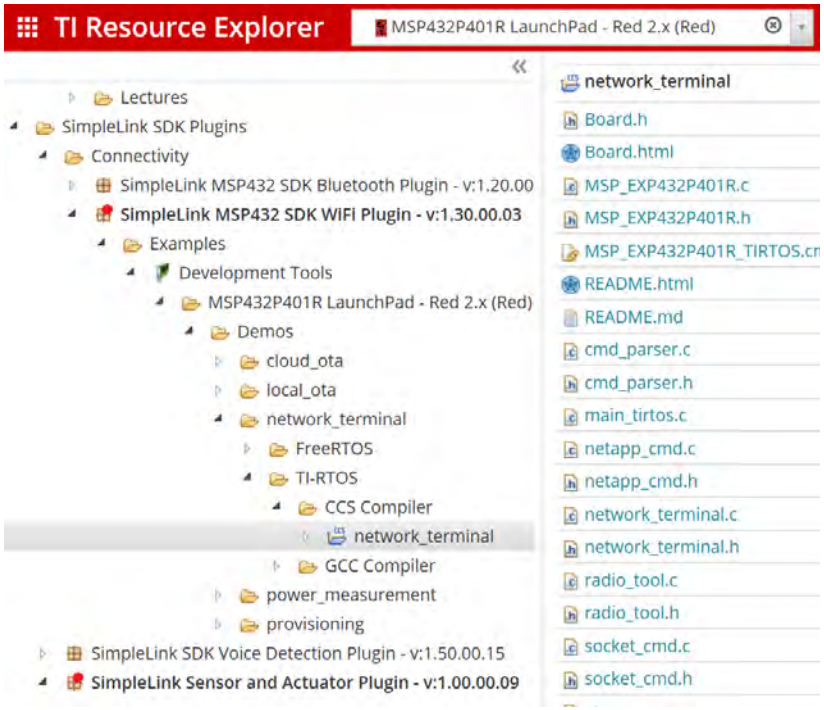


Figure 10. Resource Explorer

- Be sure you select your desired project "flavor" (TI-RTOS, Free-RTOS, CCS, GCC, etc.).
- TI-RTOS + CCS Compiler is recommended. We will be using the TI-RTOS CCS example for this lab: network_terminal_MSP_EXP432P401R_tirtos_ccs

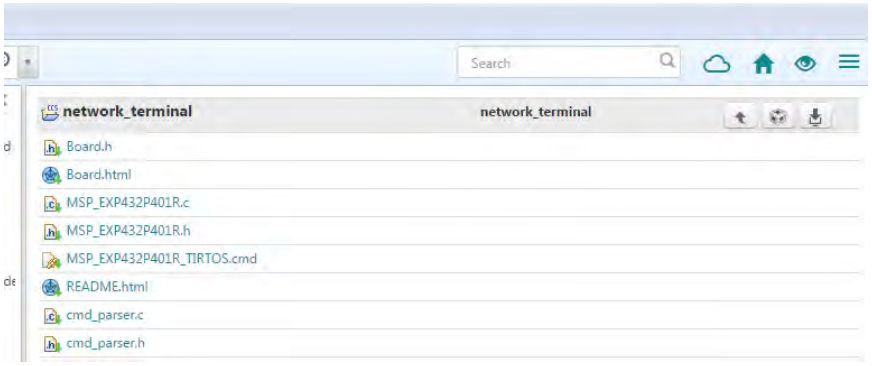


Figure 11. network_terminal CCS project files

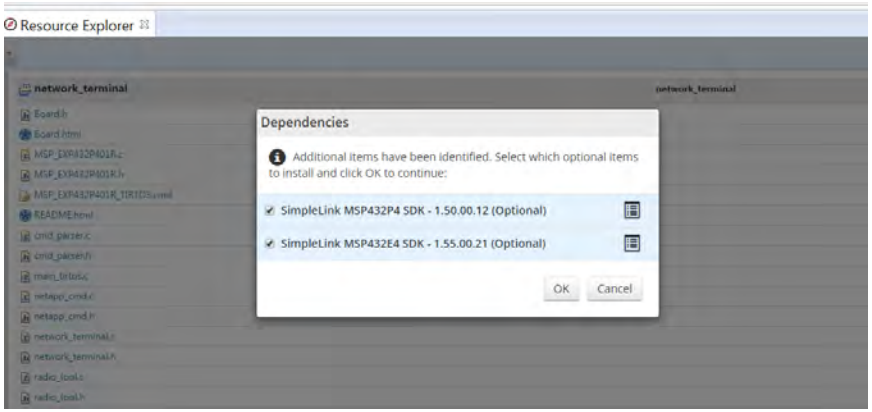


Figure 12. Install dependencies dialog box.

4. Restart CCS to install dependencies
5. Click the import to IDE button (CCS Cube icon)
6. Build the project by selecting *Build Project* from the Project menu or right-clicking the name of the project. It may take a couple minutes to build.
7. **Using CCS Debugger:** Start a Debug session by clicking the green bug in the top menu. Your MSP432 LaunchPad will need to be plugged in.
8. Open a UART terminal (or two) on your device's COM port. We want to use the XSD110 Class Application/User UART port with the following parameters:

   UART Configuration
   Baud rate: 115200
   Data: 8 bit
   Parity: None
   Stop: 1 bit
   Flow control: None

   To open in CCS go to View > Other… > Terminal > Terminal. Open a new Terminal with the above configuration

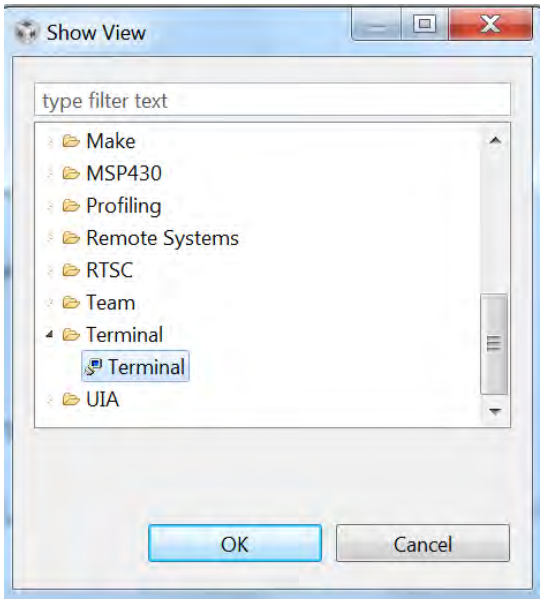


Figure 13. Show view options

Connection type: Serial Port

Click New connection. In the Launch Terminal pop up select Serial Terminal and the COM port. Confirm the configuration is the correct Baud and click ok.

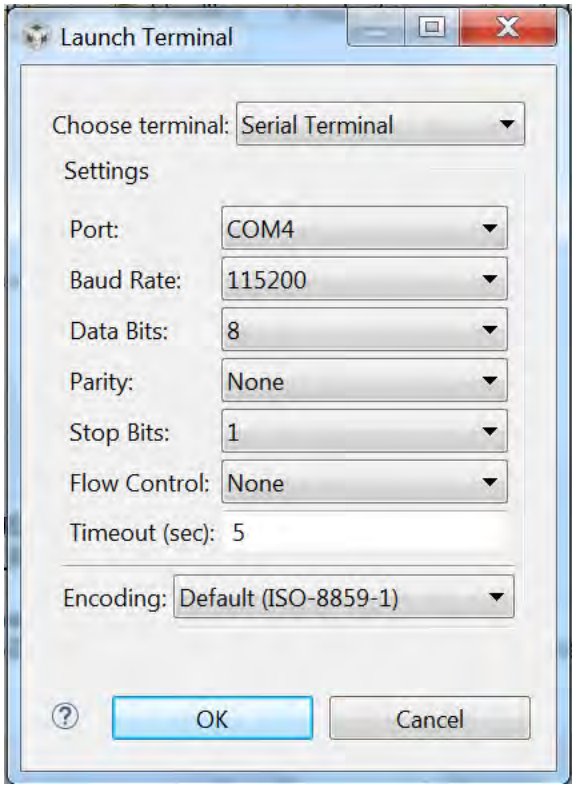


Figure 14. Serial Terminal options

9. Using the CCS debugger, click the green arrow in the top menu to start executing your code. You can now access the Wi-Fi commands by typing them in the terminal. We will explain the available commands and then use these to ping TI's website.

20.4.2 Using the Network Terminal to connect

You can type help into the terminal at any time to see a complete list of available commands.

WLAN commands

- **scan**: Retrieves scan results from network processor's (NWP) scan cache
- **setpolicy**: Defines the device's scan behavior and starts background scans
- **wlanconnect**: Connects device to an AP
- **wlan_ap_start**: Configures the device to operate in AP mode
- **createfilter**: Creates an RX filter. RX filters are a set of rules and actions imposed on each packet received from the air
- **enablefilter**: Enables all defined filters
- **disablefilter**: Disables all defined filters
- **deletefilter**: Deletes all defined filters
- **enablewowlan**: Defines a pattern-based filter, then sends host MCU to Low Power Deep Sleep (LPDS). Once the pattern filter triggers, the NWP would wake the host MCU from LPDS using host IRQ as a wake up source.
- **p2pstart**: Sets the NWP in discoverable Peer to Peer mode and connects to another visible P2P device

Socket commands

- **send**: Demonstrates opening a TCP or UDP socket, sending data in packets, and closing socket
- **recv**: Demonstrates opening a listening socket, receiving data in packets, and closing socket

NetApp commands

1. **ping**: Pings specific host name or IP, and prints statistics to terminal
2. **mdnsadvertise**: Advertises a service over mDNS
3. **mdnsquery**: Runs mDNS query for services over local LAN

Transceiver commands

- **radiotool**: Starts the Radio Tool. This allows users to run several radio-related tests for RX and TX operations

Learn more about the commands

You can learn more about any command and see an example of usage by typing [command] -help into the terminal.

To ping a website, perform the following steps in the terminal.

1) As a station, connect to an AP with internet access (a hotspot or router, for example).
2) Ping a popular website, such as google.com or ti.com.
3) Send 5 Echo-request packets with a 2 second delay interval.

```
scan -n 20
wlanconnect -s "cc3120-demo" -t WPA/WPA2 -p
"password"
ping -h www.ti.com -c 5 -i 2
```

Use your own AP's SSID and password in the wlanconnect command

You should get a response back from the website. Try entering a different website with different echo and delay interval. If you are not sure what to enter for a command you can type the command with the help flag such as "ping -help" or "wlanconnect -help"

20.4.3 Get weather and time

Now we will need to import an example to our workspace. Download the CCS Project from http://www.ti.com/lit/zip/slac767

Extract the zip file and import the project Lab20_WiFi into your workspace. The project will not build without setting up the MSP432 SDK and importing the kernel project as was described in section 20.3. If you did follow the setup procedure, then the project should build without errors but may contain warnings.

We will now look at the Lab20_WiFi project which will give a demo of connecting to a web server, openweathermap.org and nist.gov, to receive the weather and time. This program is broken down into several files with the **get_time_and_weather.h** and **get_time_and_weather.c** doing the heavy lifting. **main_tirtos.c** is a standard file used to start the RTOS kernel. **network_if.h** and **network_if.c** main job is to set up the MSP432 as a station that will connect to the local Wi-Fi router.

Open up **network_if.h** and change line 62 **#define SSID_NAME** to your router SSID name and line 66 **#define SECURITY_KEY** to your password. The most common router security will be WPA2 which is the default. If you have a password than you should leave the **SEC_TYPE_AP_MODE** on line 64 as default. If you have an open router with no password, you can change line 64

#define SEC_TYPE_AP_MODE to **SL_WLAN_SEC_TYPE_OPEN** and leave the line 64 **#define SECURITY_KEY** as a blank string.

Open **get_time_and_weather.c** and go to line 104. We will change **flashDemoConfigParams** to your Wi-Fi SSID name and password in the first and second arguments. If you have open Wi-Fi with no password again you can change the third argument to **SL_WLAN_SEC_TYPE_OPEN**, but if not you can leave as default. The fifth argument is a city. You can leave as default to get data from Dallas, TX or you can change to another city using this string. It is recommended to try to get the default Dallas data first and then change the city later.

Everything else you can leave as default and go ahead and compile and upload your program to the LaunchPad. Again open up your Terminal inside of CCS to see the UART data of the connection process, plus the current weather and time data. If you are satisfied with the output, try a different city and reflash the program to the LaunchPad. Examine the code for structure and find places where changes could be made to connect to other web servers and output data.

20.4.4 Send an email with IFTTT

Now we will interact with some easy cloud services. Let's have our robot send an email. We will use a popular aggregator service called **If This Then That**, which is a website that lets us set up rules and triggers to automate a process. For example if the weather forecast rain, send us a notification so we can prepare our umbrella.

1. Sign up for an IFTTT account at **ifttt.com** and associate it with an email address for testing your LaunchPad. Create a new applet from the interface. An applet is a logical connection between two web services supported in IFTTT.

2. Start to choose a service by clicking "this" highlighted in blue.

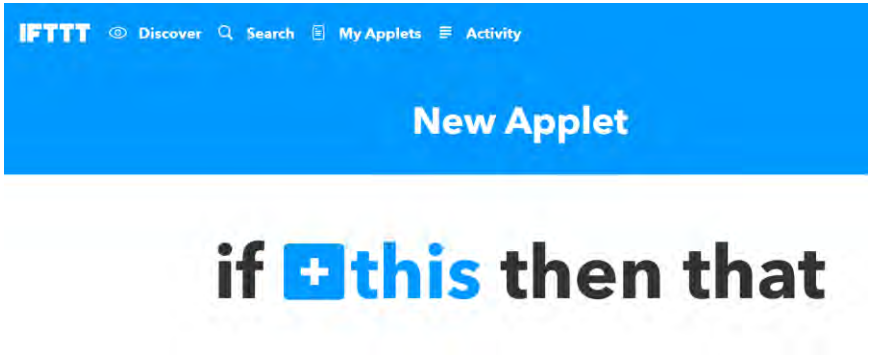


Figure 15. IFTTT new applet this

3. Choose the trigger service by typing "webhooks" and select webhooks which is also called the maker service. Enable the webhooks service if prompted.

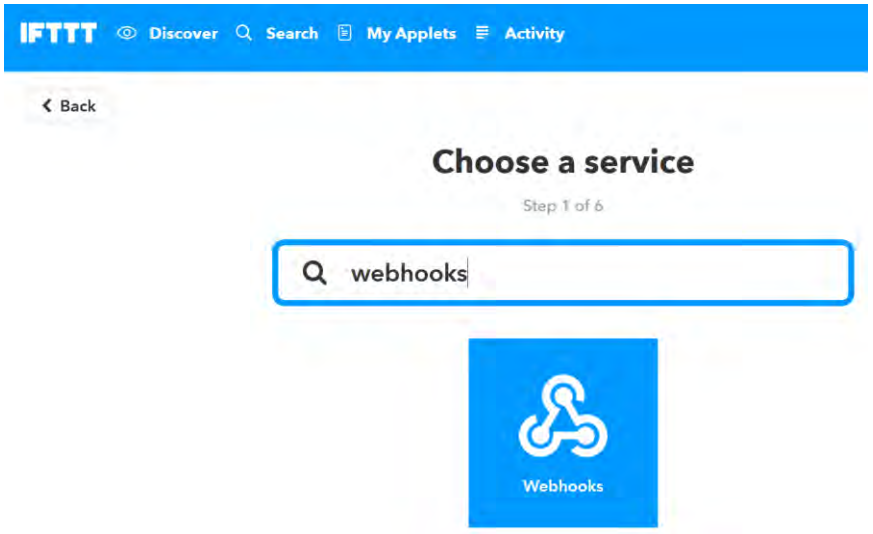


Figure 16. IFTTT choose service

4. Call the event name “button_pressed” or any you have been.

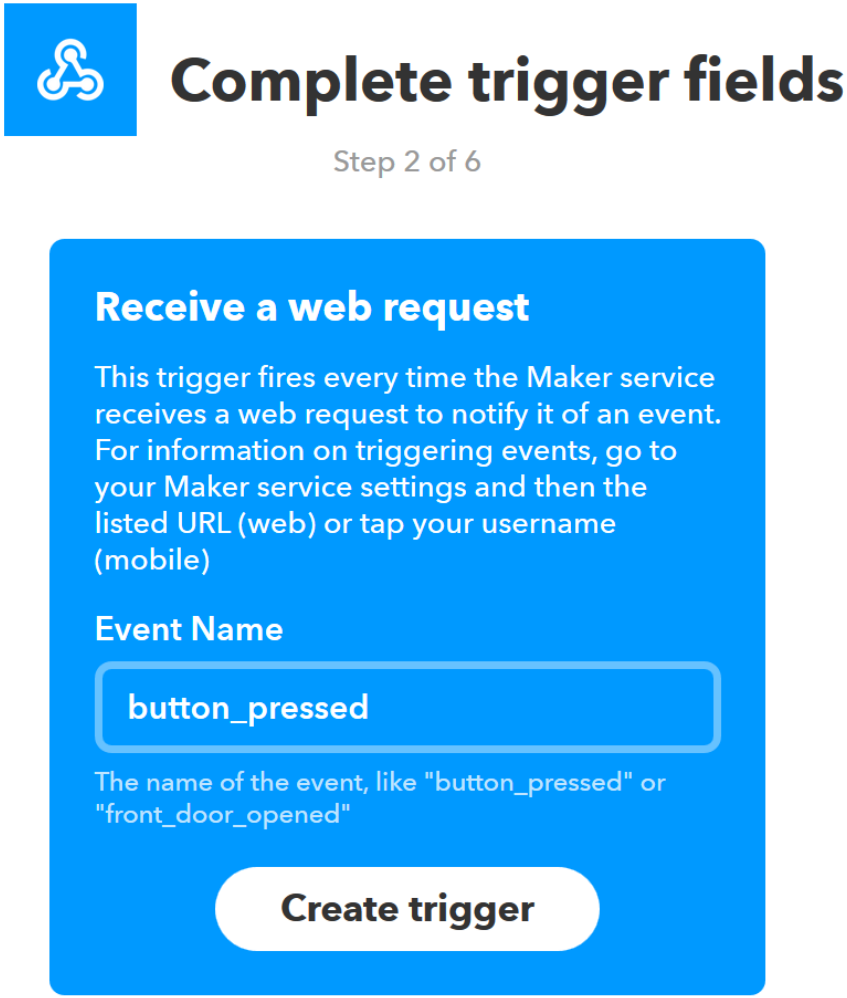


Figure 17. IFTTT complete trigger fields

5. Start to choose an action by clicking “that” highlighted in blue

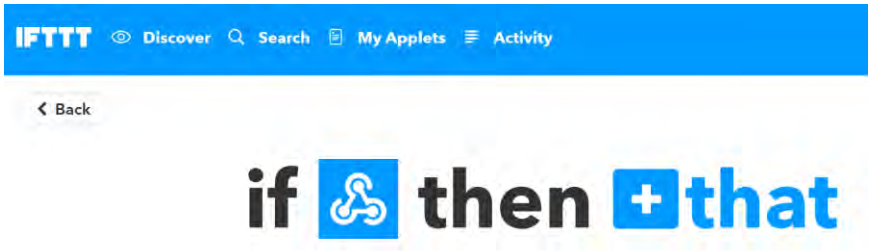


Figure 18. IFTTT new applet that

6. Choose the action service by typing “email” and select email.

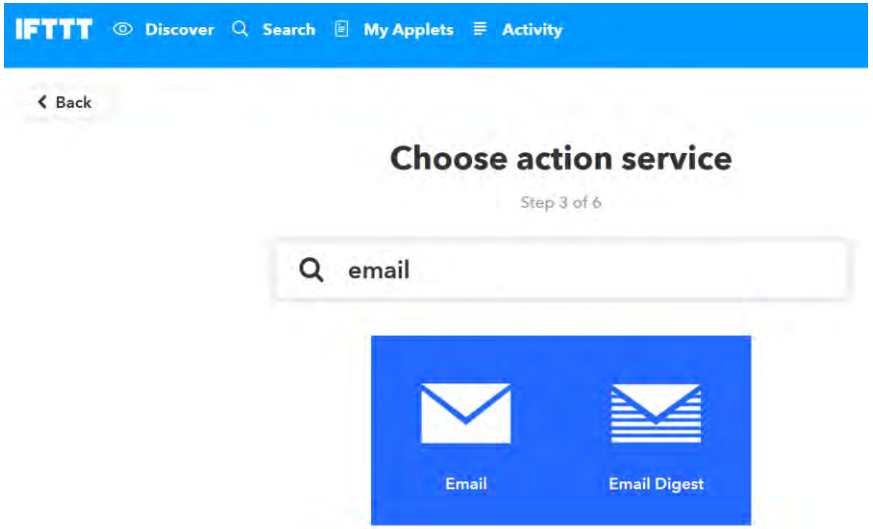


Figure 19. IFTTT choose action service

7. Choose an action “send me an email”

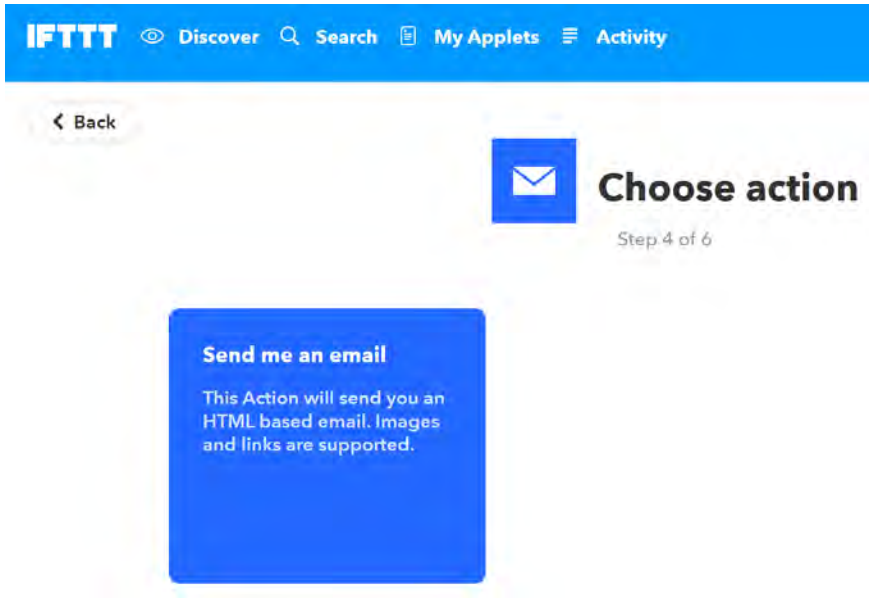


Figure 20. IFTTT choose action email

8. Set your subject to “MSP432 LaunchPad Email” and leave the body with the default values.

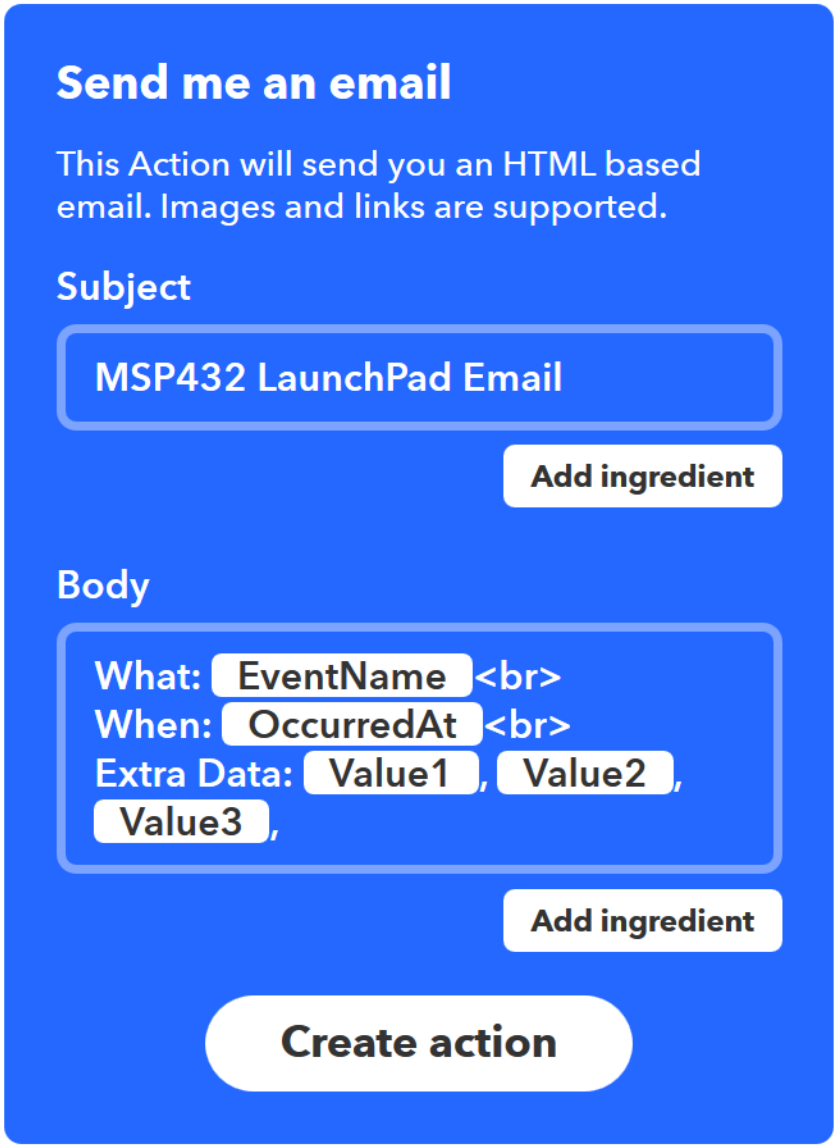


Figure 21. IFTTT action setup

9. Click “Create action” and then your applet is complete.

10. Now we need to go into the settings of the webhooks service. Go into the settings from my applets menu.

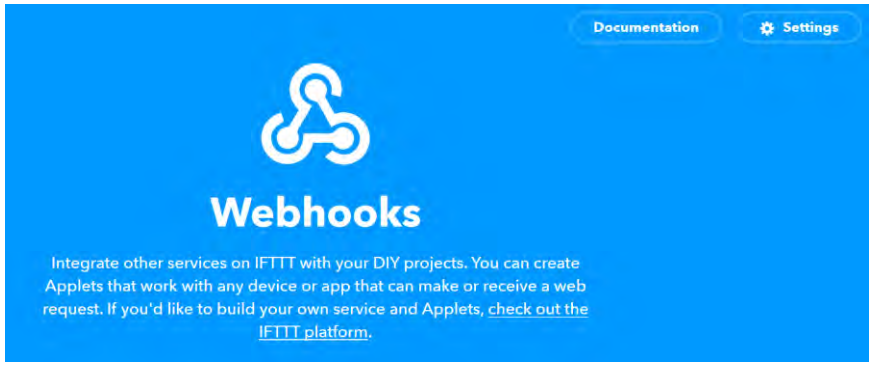


Figure 22. IFTTT Webhooks settings

Here you will see the URL you need to navigate to with your unique IFTTT key after https://maker.ifttt.com/user/{key}

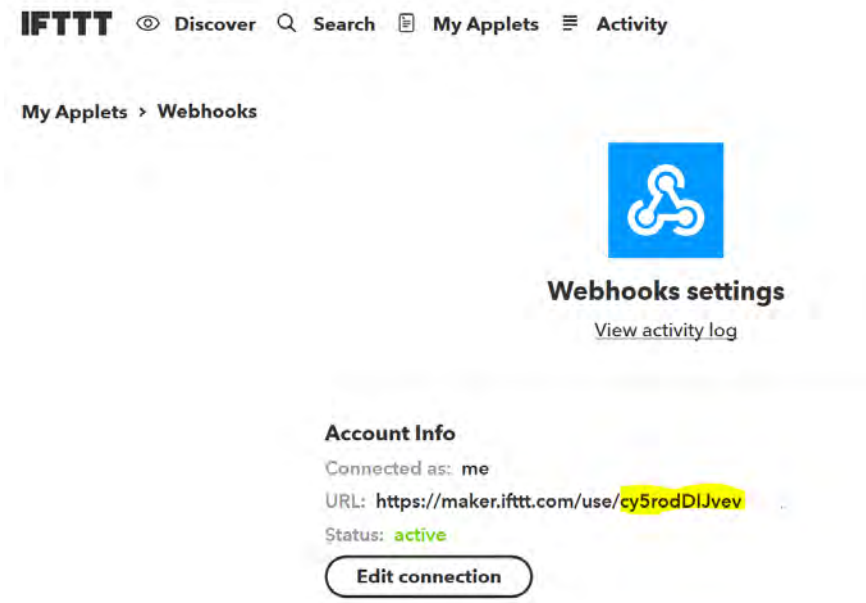


Figure 23. IFTTT webhooks account information

11. Go to that URL and it will give you instructions on how to use the service.

To send an email through IFTTT we will need to have our LaunchPad ping https://maker.ifttt.com/trigger/{event}/with/key/{key} where event is “button_pressed” and key is your IFTTT key.
https://maker.ifttt.com/trigger/button_pressed/with/key/{key}

12. You can try it out in your web browser first to verify the email sends correctly.

13. Now we will need to set up our CCS code.
Make a copy of **Lab20_WiFi** in your workspace and name it **Lab20_IFTTT**. Look into the **get_time_and_weather.c** and change your server and GET request variables to reflect the IFTTT url. Once you’ve made changes save and upload the code and utilize your debugging skills to get the project converted to the new web service. You are looking in this first stage to send the single query to IFTTT on power up just like the previous example.

The final challenge is to tie the email to a button press. Whenever a button is pressed, fire an interrupt event and trigger the IFTTT email.

Lab: Wi-Fi

20.5 Troubleshooting

The Wi-Fi can't connect to the router:

- Check all the connections between LaunchPad and the BoosterPack and make sure the pins are lined up.
- Make sure the LaunchPad is connected via USB and powered on
- Make sure the Wi-Fi BoosterPack power LED is on when connected to LaunchPad.
- Verify the SSID and password of the router you are connecting to in the code
- Make sure the router does not have a splash screen for logging in. The CC3120 is not able to know what to do with that.

Cloud issues:

- Make sure the router has an internet connection

20.6 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.

- What does it mean that this interface is serial? Why is serial important?
- What does it mean that this interface is synchronous? Why is synchronous important?

20.7 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,

- Connect your robot to the Dweet.io service and then use the visualization tool freeboard.io to display the data coming from Dweets
- Connect your robot to the Temboo service (www.temboo.com)
- Connect your robot to the Blynk service to enable Wi-Fi mobile app (www.blynk.cc)
- Connect your robot to move based on changes in the stock market
- Set your robot as an AP to transmit diagnostics and sensor data to a connected web client

20.8 Which modules are next?

Modules 1-20 have introduced the basics of the microcontroller and advanced functionality to add to the robot. You should have most of the ground work to complete the robot challenge. Additional supplemental modules are available for study on other techniques and concepts.

Module 21) robot challenges

20.9 Things you should have learned

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

- Understand basic procedure of Wi-Fi connections
- Utilize the SimpleLink SDK to get started quickly with Wi-Fi development

g mode in CCS

Module 21

Robot Challenges – Solve the maze

Robot Challenges

Educational Objectives:

The purpose of the robot challenge is to combine previous modules into a system that solves a complex task. A line-following challenge can be attempted after module 13. You can attempt the simple maze challenge after finishing module 15. More advanced challenges will require additional modules. Solutions to the labs provide components (hardware and software) for the challenge. In the challenge, you perform system-level design with the components developed in previous modules.

Good to Know: This challenge allows you to perform duties typical of the engineering profession. The first typical engineering duty is expansion or modification. In other words, given a system that works, how might we reuse the solution to solve a similar but slightly different problem? For example, you interfaced 6 switches in lab 10, and now you might wish to add a 7th switch. The duty would then be to rework the 6-switch solution so it now allows 7 switches. The second engineering duty is integration. In other words, given two systems that work, how might we combine the two systems to create a more complex system?

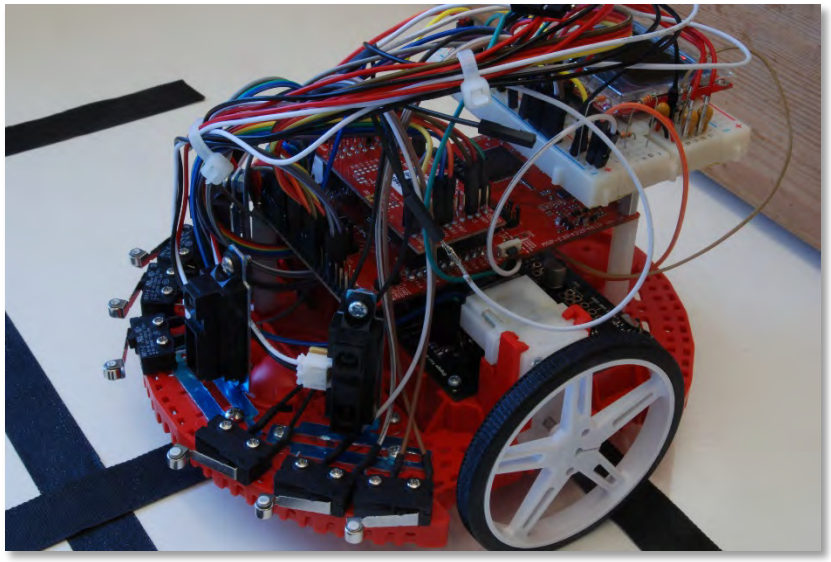


Figure 1. System integration to solve a complex task.

1. Getting Started

1.1. Software Starter Projects

Recall that your low-level drivers are located in the **inc** folder. This approach allows you to reuse code from one lab for the next lab. Together with your solutions to the other labs, look at these two projects, which combine your previous modules into a single system:
Jacki (an empty project up through module 17)
JackiFSM (empty project up through module 13)

If you are attempting a challenge without completing the individual labs beforehand, there are two projects that embed object-code solutions to the low-level drivers. Since the low-level drivers are provided in object code form, this approach will not allow you to learn how it works.
Competition (a starter project)
Competition_BLE (a starter project with BLE)

1.2. Student Resources (Links)

MSP432 Technical Reference Manual (SLAU356)
Meet the MSP432 LaunchPad (SLAU596)
MSP432 LaunchPad User's Guide (SLAU597)
QTR-8x.pdf, line sensor datasheet
GP2Y0A21YK0F_IR_Distance_Sensor.pdf, datasheet
Polulu_BumpSwitch_1404.png, mechanical drawing of switch
MotorDriverPowerDistribution.pdf Data sheet for power board
PololuRomiChassisUsersGuide.pdf How to build the robot
drv8838.pdf Data sheet for the H-bridge driver

1.3. Reading Materials

Refer to the book readings for each of the modules with which you combine to make your robot.

1.4. Components needed for this lab

Refer to the components needed for each of the modules with which you combine to make your robot. Refer to the construction guide for a complete description of how to build the robot.

1.5. Lab equipment needed

Voltmeter
Oscilloscope (one or two channels at least 10 kHz sampling)
Logic Analyzer (4 channels at least 10 kHz sampling

2. Design Requirements for Basic Challenges

Challenge: Feel free to adapt/combine these ideas into a problem that the integrated robotic system can solve.

2.1. Design Challenge 1: Line following

If your robot has a line sensor and the bump sensors, you could create a line follower that races along a line, and uses the walls to detect when it has strayed far from the line, see Figures 2 and 3. Knowing which bump sensors were triggered tells you the angle it hit the wall. Knowing the angle allows you to back up, turn around and head back to the center of the room, searching for the line again.

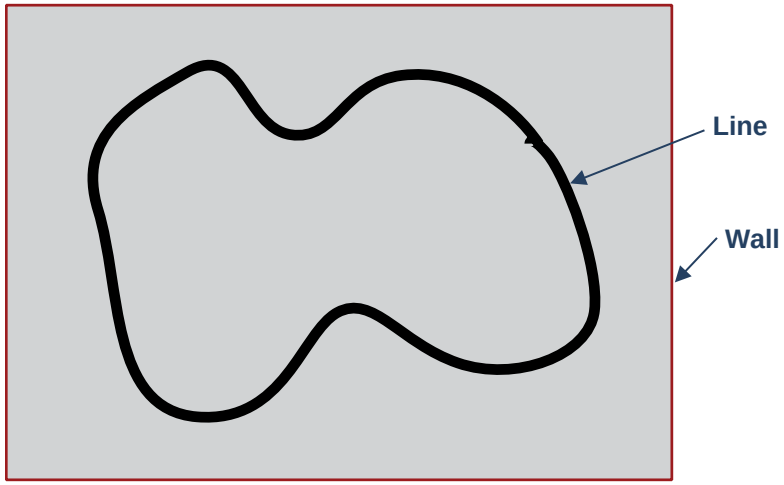


Figure 2. Create a robot explorer that follows a line.

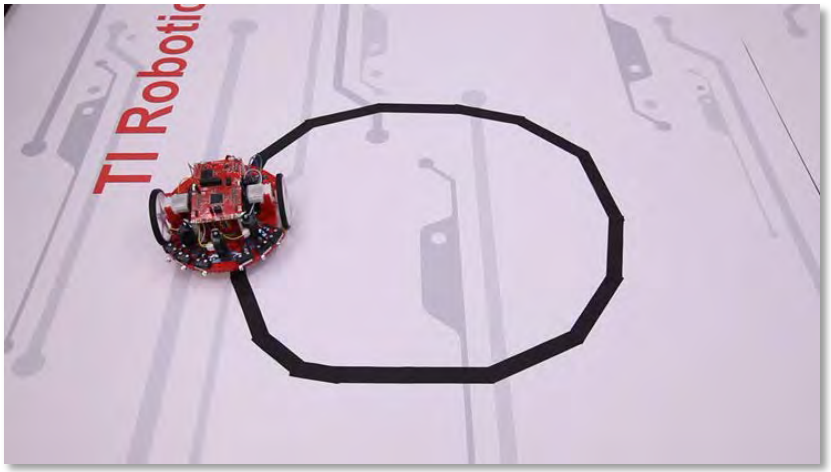


Figure 3. Create a robot explorer that follows a line.

To solve this challenge, the minimum set of modules you need are:

- *Module 6: GPIO to interface the line sensor*
- *Module 7: FSM as an appropriate approach for line following*
- Module 10: Use SysTick periodic interrupts for the line sensor
- Modules 12+13: Motors and PWM for the robot

Module 14, which is optional, could be used to interface the bump sensors.

2.2. Design Challenge 2: Line-following to search for treasure

If your robot has a line sensor and the bump sensors, you could create a maze solver that searches for treasure, see Figure 4. Like challenge 1, it uses the walls to detect when it has strayed far from the line. This option will require a high-level maze solving strategy. You will need a mechanism to determine when the treasure has been reached. Possibilities for detecting the treasure include:

- Detecting a special pattern, like 0101010;
- Making the treasure taller than the wall and placing some switches at a height higher than the wall so there switches get triggered only at the treasure.

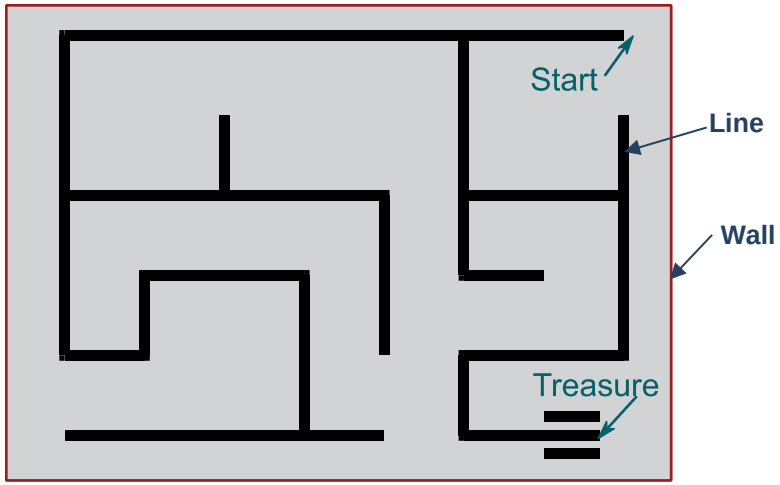


Figure 4. Create a robot explorer that finds its way out of a maze, using just the line sensors and bump sensors.

To solve this challenge, the minimum set of modules you need are:

- *Module 6: GPIO to interface the line sensor*
- *Module 7: FSM as an appropriate approach for line following*
- Module 10: Use SysTick periodic interrupts for the line sensor
- Modules 12+13: Motors and PWM for the robot

Module 14, which is optional, could be used to interface the bump sensors.

2.3. Design Challenge 3: Bump and run exploration to find treasure

If your robot does not have a line sensor, you could create a maze solver that searches for treasure using just the bump sensors for guidance, see Figure 5. This robot must bump into the wall as it feels its way around the course. The walls are at a height that can be sensed by some of the bumper switches. The treasure is taller than the walls, and some switches are placed at this higher location, allowing the robot to distinguish between wall and treasure. When the robot has found the treasure, it should stop and flash its LEDs to signify success.

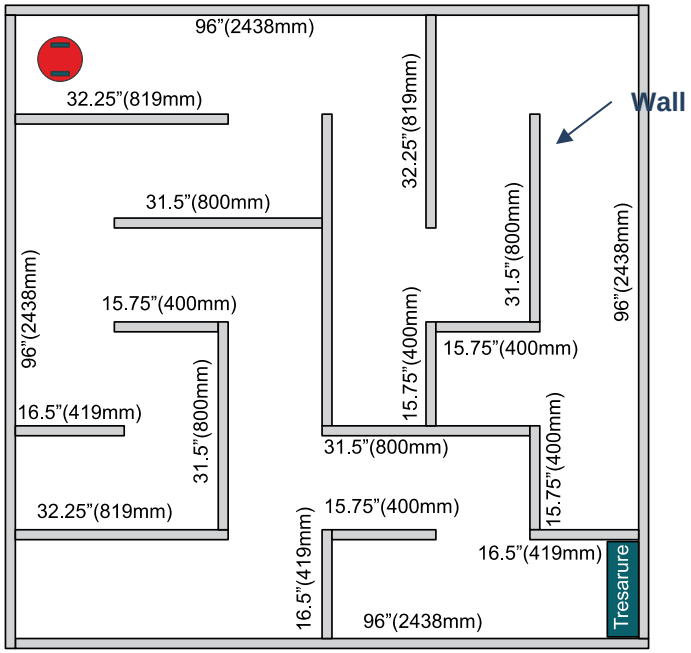


Figure 5. Create a robot explorer that finds a treasure within the maze. The treasure is taller than the walls.

To solve this challenge, the minimum set of modules you need are:

- *Module 7: FSM as an appropriate approach for searching*
- Modules 12+13: Motors and PWM for the robot

Module 14, which is optional, could be used to interface the bump sensors.

3. Design Requirements for Advanced Challenges

3.1. Design Challenge 5: Maze exploration

If your robot has the IR sensors but not the line sensor, you could create a maze solver that searches for treasure using just the IR distance sensors for guidance. The same maze structure shown in Figure 6 could be used. This robot should not bump into the wall. Rather, it uses the IR sensors to avoid the walls. The walls are at a height that can be sensed by some of the bumper switches. The treasure is taller than the walls, and other switches are placed at this higher location, allowing the robot to distinguish between wall and treasure. When the robot has found the treasure, it should stop and flash its LEDs to signify success.

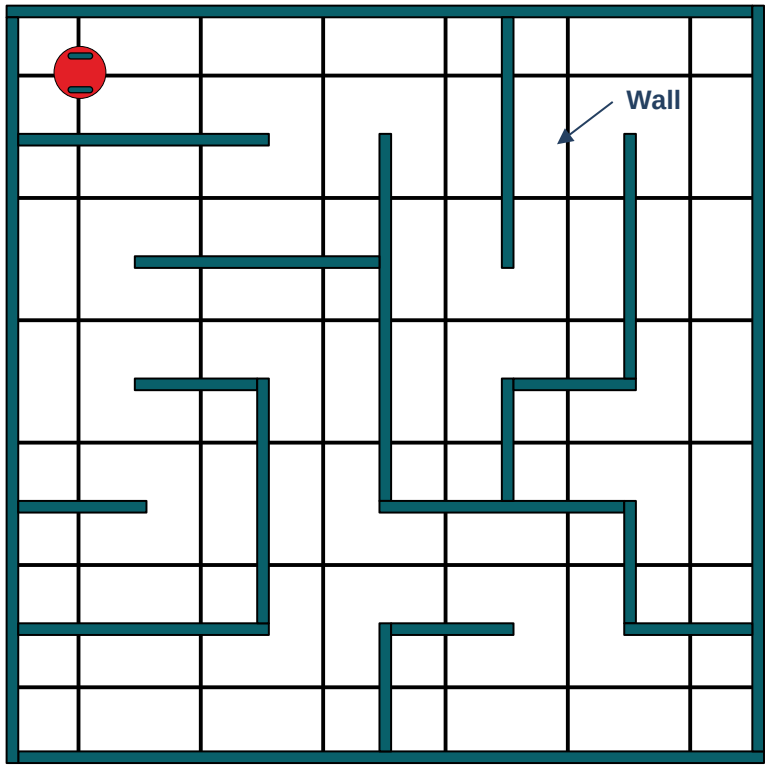


Figure 7. Create a robot explorer that finds a treasure within a maze. The treasure is taller than the walls.

To solve this challenge, the minimum set of modules you need are:
Module 4: Pattern recognition to help guess which way to go
Module 6: GPIO to interface the line sensor
Module 7: FSM as an appropriate approach for line following
Module 10: Debug to use SysTick periodic interrupts for the line sensor
Modules 12+13: Motors and PWM for the robot
Module 14: Edge triggered interrupts for the collision sensors
Module 15: IR distance sensors used to sense the walls.
Modules 16+17: Tachometer and control system (optional)

Robot Challenges

3.2. Design Challenge 6: Autonomous racing

If your robot has the IR sensors but not the line sensor, you could create an explorer robot using just the IR distance sensors for guidance. The goal is to travel around the world as quickly as possible, see Figures 8 and 9. The robot uses the IR sensors to avoid the walls and the other robots. The walls are at a height that can be sensed by the bumper switches. You can race one at a time or you can race multiple robots at the same time.

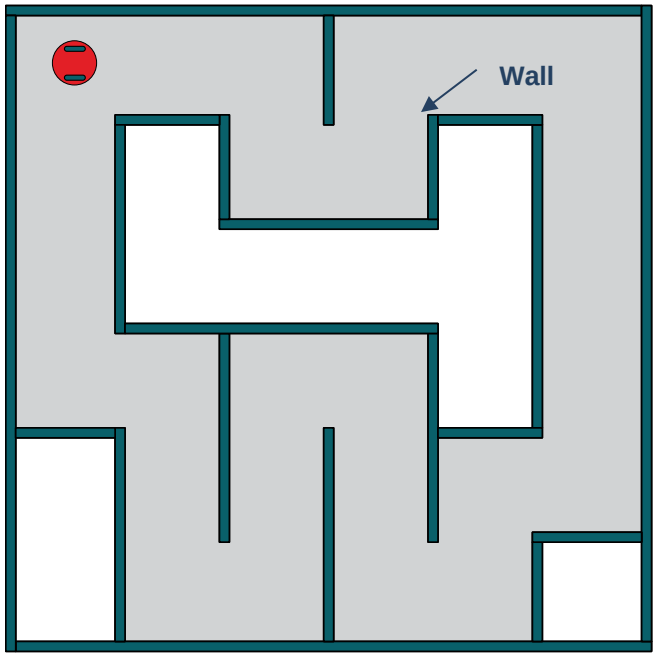


Figure 8. Create a robot explorer that races around a track using bump sensors and IR sensor.

Figure 9. The track is wide enough that the IR distance sensors are monotonic.

To solve this challenge, the minimum set of modules you need are:
Module 4: Pattern recognition to help guess which way to go
Module 7: FSM as an appropriate approach high-level design
Modules 12+13: Motors and PWM for the robot
Module 14: Edge triggered interrupts for the collision sensors
Module 15: IR distance sensors used to sense the walls.
Modules 16+17: Tachometer and control system (optional)

Robot Challenges

3.3. Design Challenge 7: Autonomous racing with sensor integration

If your robot has the IR sensors and the line sensor, you could create an explorer robot using both the line sensor and the IR distance sensors for guidance. The difficulty will be to integrate data from both types of sensors. The goal is to travel around the world as quickly as possible, see Figures 10 and 11. The robot uses the IR sensors to avoid the walls and the other robots. It could use the line sensor to orient itself relative to the walls. The walls are at a height that can be sensed by the bumper switches. Because of the width of the robot compared to the size of the track, this challenge was meant to race one robot at the same time.

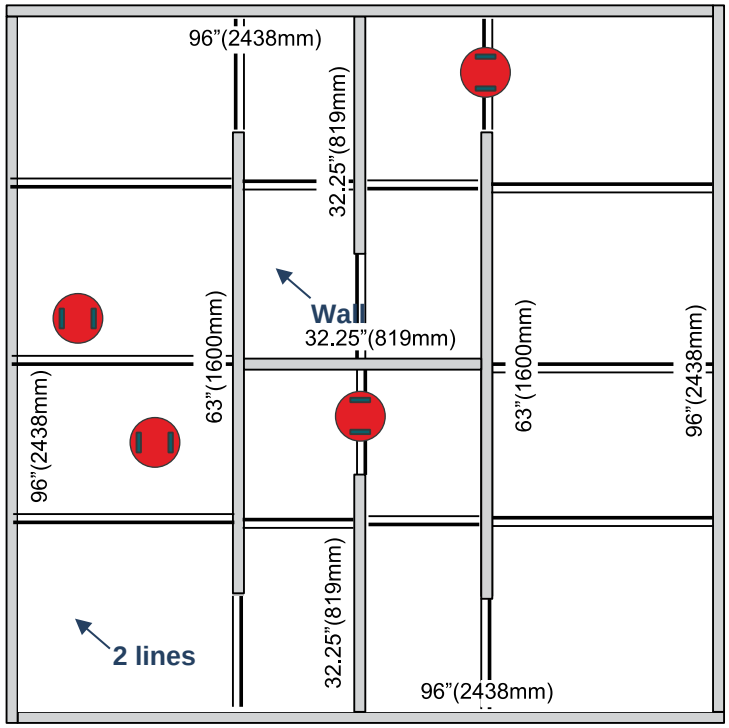


Figure 10. Create a robot explorer that races around a track with bump sensors, line sensors, and IR sensors. Multiple robots can race at the same time. The lines are perpendicular to walls. Each line marking is has two lines (thick and thin) so robot can measure angle and direction.

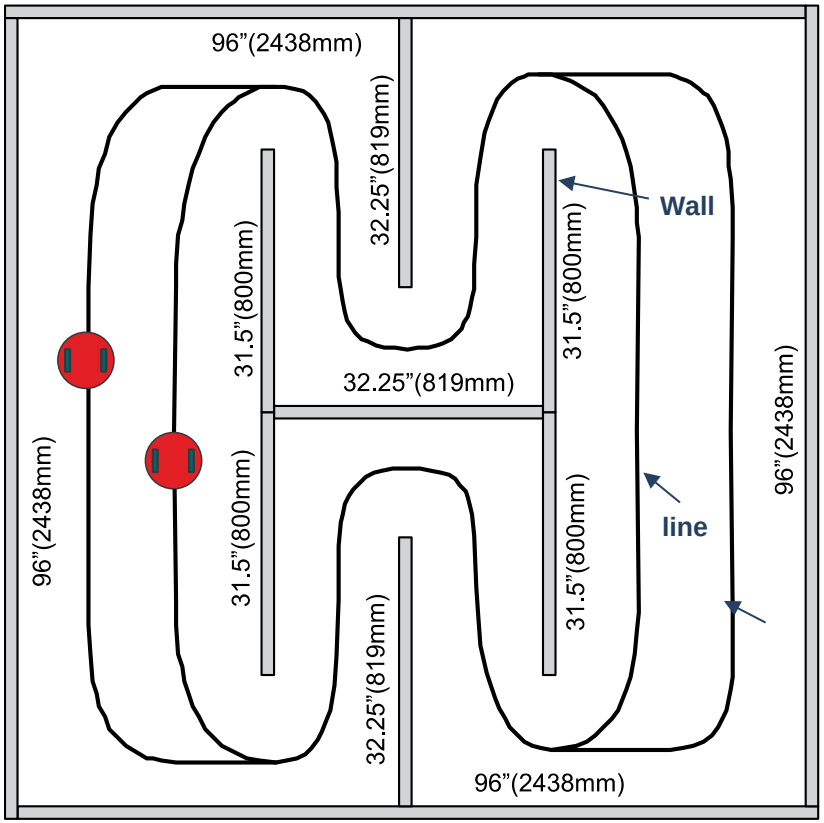


Figure 11. Create a robot explorer that races around a track with bump sensors and line sensors.

To solve this challenge, the minimum set of modules you need are:
Module 4: Pattern recognition to help guess which way to go
Module 6: GPIO to interface the line sensor
Module 7: FSM as an appropriate approach high-level design
Modules 12+13: Motors and PWM for the robot
Module 14: Edge triggered interrupts for the collision sensors
Module 15: IR distance sensors used to sense the walls.
Modules 16+17: Tachometer and control system (optional)

Robot Challenges

4. Experiment set-up

For each module you choose to deploy refer back to that module to configure and hardware and software needed. To build the arena, the walls need to be heavy enough so the robot does not push it over.

Construction approach 1: One very flexible approach to building the arena is to use individual 4 by 4 pieces of wood, see Figure 12. Wood this size is heavy enough to be placed on the floor without fastening the pieces together.

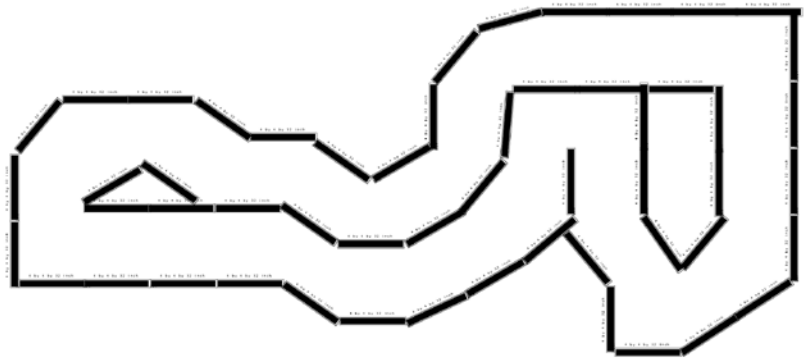


Figure 12. The Formula 1 race track at Austin Texas can be built with 54 pieces of 4 by 4 wood.

Construction approach 2: Another approach to building the arena is to use individual 2 by 4 pieces of wood cut to specification. The mazes shown in Figures 2 through 11can be built with pieces of 2 by 4. Wood this size is not heavy enough to be placed on the floor without fastening the pieces together. So these pieces will need to be screwed or hammered together, which is why the designs all have 90 degree angles.

The robot is 6.43 inches (163 mm) round. The arena is 8 foot by 8 foot. You can think of the space as 36 individual rooms, each room is 15.75 inches square (400mm). Walls are 1.5 in wide (so effective room size or door clearance is 14.25 inches, 362 mm).

Pieces need to construct the mazes shown in Figure 5, 6, and 7 are listed in Table 1.

count	in	mm
4	96	2438.4
4	15.75	400.05
3	16.5	419.1
4	31.5	800.1
3	32.25	819.15
1	47.25	1200.15

Table 1. Materials needed to build the maze shown in Figures 5-7.

Multiple mazes by flipping and rotating (start and end in different corners), as shown in Figure 13.

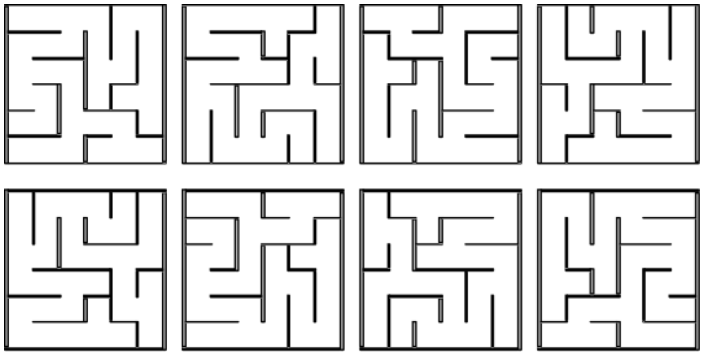


Figure 13. Multiple mazes can be made by rotating and flipping.

Figures 9, 10, and 11 define race tracks. Racing is a fun class activity and really rewards good design. Table 2 shows the pieces needed to construct these race tracks.

count	in	mm
4	96	2438.4
2	63	1600
3	32.25	820

Table 2. Materials needed to build the maze shown in Figures 9-11.

5. System Development Plan

5.1. Top-down design

The entire curriculum was based on the bottom-up approach, starting with simple components. After you learned how a component operates, it was abstracted, creating a set of functions you need to use it. In the challenge, however, we will take a top-down approach. There are five phases of the project:

1. Analysis (requirements, specifications, constraints)
2. High-level design (strategies, data flow graph, algorithms, abstractions)
3. Low-level design (call graph, header files, data structures, flow charts, how will it be tested?)
4. Implementation (modularity, concurrent development)
5. Testing (bottom-up testing, control and observability)

Strategy. You will begin with developing an overall strategy. An important decision is which sensors to use and how you plan to use them. You will need a plan to balance speed with accuracy.

The finite state machine is one approach to consider for implementing line following. The FSM from Lab 6 had only 2 inputs, see Figure 14. Your solution had more states, but it maintained the 2 input/ 2 output structure. One option is to distill the line center data to create the 2 inputs as envisioned when you solved Lab 6. Furthermore you can use the FSM output values to set the duty cycle for each motor.

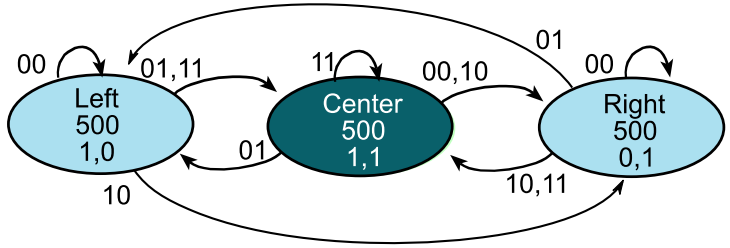


Figure 14. Moore FSM state graph to implement line following. The time in each state is shown in 1ms units.

If you review your Lab 6 results you will find the line sensor output is not a continuous variable, but rather can take on only a finite number of values. Furthermore if you distill the line sensor data into eight distinct classifications, you can use these eight discrete input possibilities to drive a line following robot similar to Lab 6. Each state in the FSM has 8 next-state arrows.

- Lost
- Way off to left
- Off to left
- Little bit left
- Centered
- Little bit right
- Off to right
- Way off to the right

A Fuzzy Logic controller could also be used to follow the line.

Module 17 introduced a number of control algorithms you could use to move in a straight line or follow along a side wall.

5.2. System integration

Once an overall plan is developed you need to integrate components from the other labs to create a system that performs an overall task. Consider how priority is assigned. Consider how the different software threads are integrated into one system. For example, you could define four threads

- Periodic SysTick interrupts to measure the line sensor
- Periodic Timer A1 interrupts to run the high-level strategy
- Edge triggered interrupts for collisions
- Main program for debugging and low priority tasks

Recall that the PWM outputs to the motor do not require software action to operate. Once configured the Timer_A hardware automatically produces the PWM outputs. Software needs to execute only when the direction or duty cycle are to be changed.

Next, you need a mechanism to pass data between threads. Semaphores, mailboxes, and FIFO queues are appropriate for this project. For example, the ISR might just stop the motors and set the Collision mailbox. Subsequently, the Timer A1 periodic thread can handle the collision, backing and turning as needed.

5.3. Testing

Effective debugging involves *control* and *observability*. Four strategies for controlling what your robot does (without going through the edit, compile, download, debug cycle) are

- Connect the robot via a long USB cable and run the debugger
- Use switches on the robot to select various modes/strategies
- Connect the robot via a long USB cable and implement an interpreter via the UART on the MSP432 and a terminal emulation program running on the PC. This way you can send commands and observe responses (Lab 18).
- Interface the CC2650 and interact with the robot over BLE (Lab 19)

Four strategies for observing internal parameters within your robot as is attempts the challenge are

- Connect the robot via a long USB cable and run the debugger
- Output strategic parameters to the LCD (Lab 11)
- Connect the robot via a long USB cable and output debugging information to the UART. Observe the information using a terminal emulation program running on the PC (Lab 18).
- Interface the CC2650 and interact with the robot over BLE (Lab 19)

5.4. Team management

5.5.

"A team is a small number of people with complementary skills who are committed to a common purpose, performance goals, and approach for which they are mutually accountable."

1. (Katzenbach, J.R. & Smith, D.K. (1993). The Wisdom of Teams: Creating the High-performance Organization. Boston: Harvard Business School.)
2. Decker, Philip, J. (1996) "Characteristics of an Effective Team," http://www.cl.uh.edu/bpa/hadm/HADM_5731/ppt_presentations/29teams
3. Breslow, L. (1998). Teaching Teamwork Skills, Part 2. Teach Talk, X, 5. http://web.mit.edu/tll/published/teamwork2.htm. 13 May 2003.
4. Building Blocks for Teams, (Website). Penn State University, http://tlt.its.psu.edu/suggestions/teams/student/index.html

Respect. Team effectiveness requires mutual respect. It is expected that team members will disagree. Managing conflict will be easier from a position of mutual respect. When arguing with your teammates, please consider strongly the possibility that you might be the one who is wrong.

Communication. Team effectiveness requires effective and constant communication. Create a channel (googledoc, slack, signal etc.) where thoughts and interactions can be recorded. Listen attentively and respect your teammates. Ask lots of questions. Give constructive feedback. Present your ideas forcefully, but keep an open mind. Restate the original idea to be sure it's understood. Critique the idea, not the person. Be courteous. Be aware of body language and tone. Meetings don't need to be a death march, use humor effectively. Laugh with someone, do not laugh at someone.

Leadership. A leader is responsible for 1) calling meetings; 2) finding a mutually agreeable time and place to meet; 3) setting a meeting agenda; 4) facilitating the meeting; 5) monitoring progress against the plan; and 6) identifying problem areas that need action. The leader is not "the boss". The team needs to agree on decisions and directions. Compromise is essential.

Effective Meetings. Before the meeting, name someone to be the facilitator, create an agenda and send it to all team members. Set a time limit for the meeting. During the meeting, if issues emerge that are not on the agenda, the facilitator should: ask the team if this should be discussed now, or table the issues for the end of the meeting. During the meeting, keep a list of decisions and actions items, keep to the time commitment, create an agenda for next meeting and agree on time and place. After the meeting, send out a brief summary of a list of action items and those responsible for those actions.

Robot Challenges

Brain storming. Select someone to be the recorder. Invite everyone to give their ideas and input. Write down all ideas without criticism or discussion. Avoid being judgmental of others' ideas. Try to look at all sides of an idea. Listen attentively and treat your teammates' opinions with respect. Try to encourage the widest range of new ideas. Everyone should participate. Don't stop the idea session too soon. After complete list is generated, return for discussion/analysis. Carefully select the best approaches or ideas from the list. Try to remove your ego from the discussions. Don't take the rejection of your ideas personally.

Code repository. Use a code repository, with version control, so multiple members of the team can work simultaneously.

Fail fast. Identify the component of the project that has the most risk (least likely to work), and determine quickly if it will be feasible. Always develop a plan B when proposing a strategy that may be risky.

Be realistic, be simple. Many teams fail because they attempt a strategy that is too complex. Many teams get frustrated over the size of the project, and over thoughts that they are working alone. A simple strategy often yields acceptable results when time is constrained.

Be the team that is having the most fun!

Effective team checklist

- Define a common goal for the project.
- List tasks to be completed.
- Assign responsibility for all tasks.
- Develop a timeline and stick to it.
- Develop and post a Gantt chart for the plan
- Document key decisions and actions from all team meetings.
- Send reminders when deadlines approach.
- Send confirmation when tasks are completed.
- Collectively review the project output for quality

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources 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.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

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