TI-RSLK MAX

Texas Instruments Robotics System Learning Kit

Texas Instruments

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 and line sensors that the robot will use to explore its world, see Figures 1 and 2. The software input/output with the line sensor will be moved to the background using interrupts, creating a more efficient utilization of processor time.

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 measuring the position of the line in the background.
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.

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*)*
QTRX line sensor datasheet

10.1.3 Reading Materials

Chapter 10, "Embedded Systems: Introduction to Robotics"

Note: We chose the sampling rate of the sensors to be 100 Hz, because 100 Hz (1/10ms) is about 10 times faster than the time constant of the motors used in this robot 10 Hz (1/100ms).

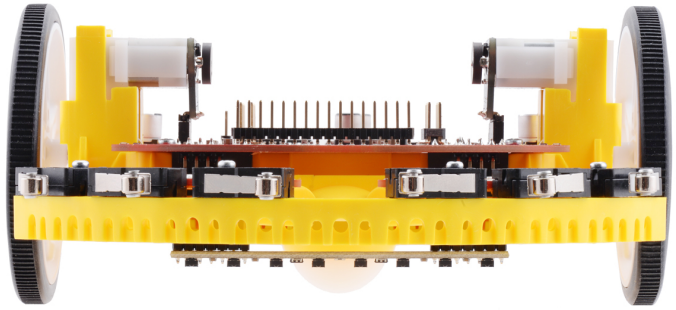

Figure 1. Line sensor positioned on the bottom of the robot (www.pololu.com).

10.1.4 Components needed for this lab

All the components needed in the lab are included in the TI-RSLK Max kit (TIRSLK-EVM kit). You can use the robot you build in lab 5. You will need black tape and batteries.

Quantity	Description	Manufacturer	Mfg P/N
1	TI-RSLK MAX kit	TI	TIRSLK-EVM

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.

.

10.2 System Design Requirements

The first goal of this lab is to implement the line sensor measurement using SysTick interrupts. The line sensor was previously interfaced in Lab 6. Here in Lab 10 you will also 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 write software to read the state of six bump sensors on the robot. The bump sensors allow the software to know if and where the robot has collided with an obstacle. Figure 2 shows the standard configuration.

Bump5 Bump4 Bump3 Bump2 Bump1 Bump0

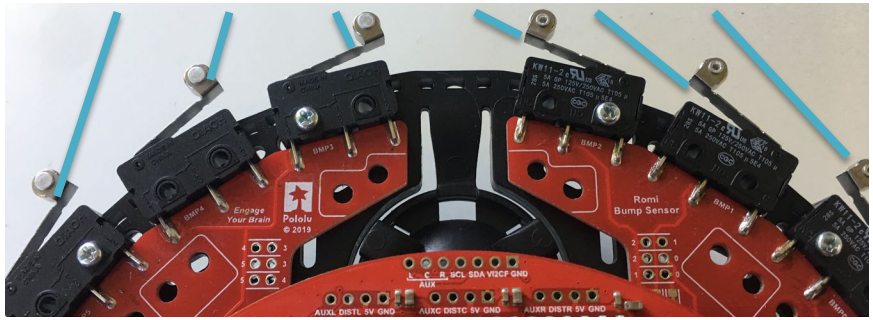

Figure 2. Bump sensors attached to the front of the robot (top view).

The third goal is to develop a minimally intrusive debugging instrument called a **dump**. We can use dumps for debugging as a substitute for print statements in real time systems like the robot. 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 line sensor 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 much 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.

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 can use the robot you assembled in Lab 5. You interfaced the QTRX line sensor in the Lab of Module 6. You also learned how to interface switches in Lab 8. In this lab you will use bump switches attached to the front of the robot and write software to read the status of the six switches. Figure 2 shows one standard placement for six sensors.

Warnings: ***TI MSP432 pins are not 5V tolerant; you must power the line sensor and bump sensors with +3.3V.***
Please also ensure the +5V jumper on the MSP432 LaunchPad is disconnected or removed. Not removing this jumper will cause permanent damage to the LaunchPad and the TI-RSLK chassis board.

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 and P9.2 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 and P9.2) 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()**.

We recommend you observe the analog waveform on the Port 7 inputs. Depending on the reflectivity of the line and background, you may have to increase or decrease the time between calling **Reflectance_Start()** and **Reflectance_End()**. You can adjust this time simply by changing the rate of the periodic interrupt.

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.

10.4.2 Interfacing the bump sensors

You can implement positive logic or negative logic. You can implement internal resistors or external resistors. The standard configuration uses negative logic with internal pull-up resistors.

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;
  }
}
```

Lab: Debugging Real Time Systems

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.

```
#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 should 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?
- The standard RSLK kit uses negative logic for the bump switches. Does it matter if we use positive or negative logic when interfacing switches?
- 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

ti.com/rslk

IMPORTANT NOTICE AND DISCLAIMER

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

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

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

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