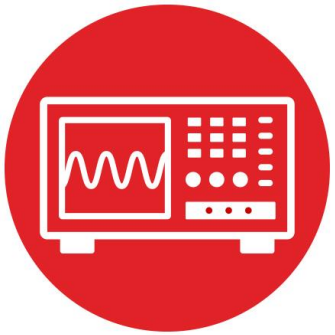

# Module 10

## 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.

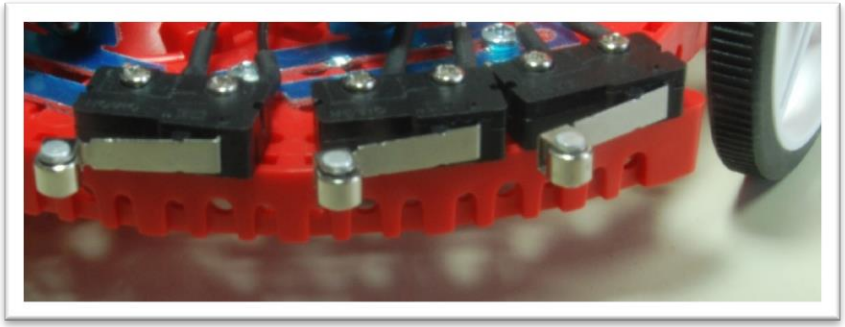

*Figure 1. Bump sensors positioned at the front of the robot.*

## 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.

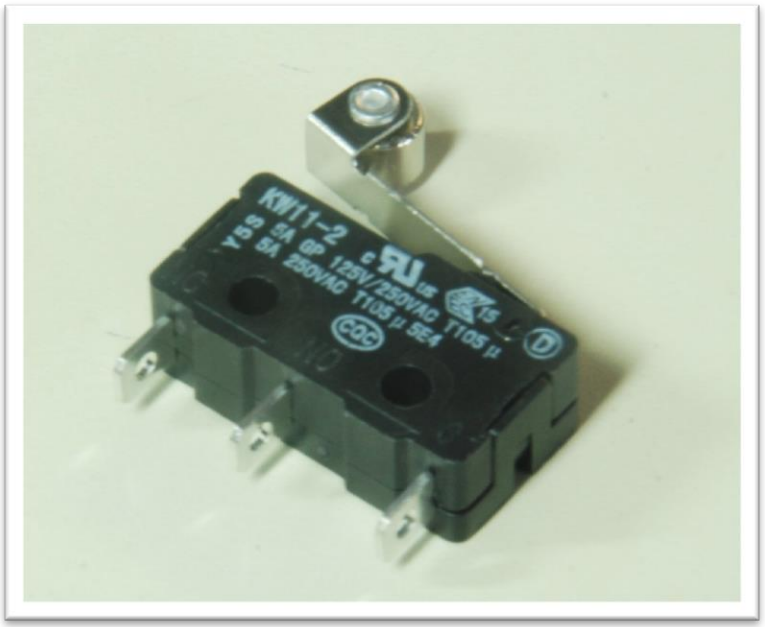


*Figure 2. Momentary contact switches allow for collision detection.*

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.*

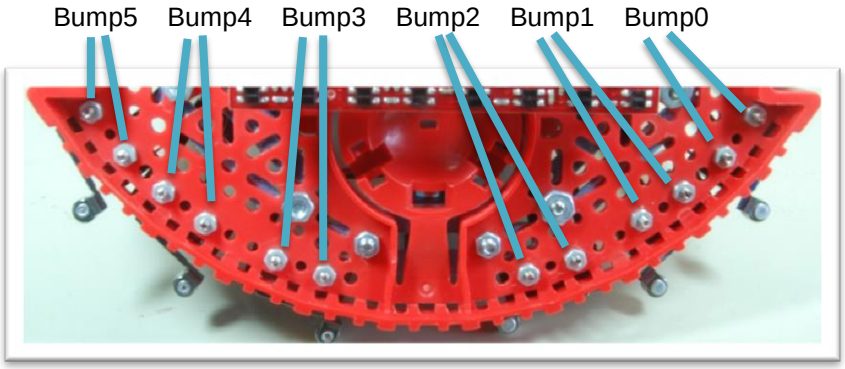


*Figure 3. Bump sensors attached to the front of the robot (bottom view).*

## 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

## 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