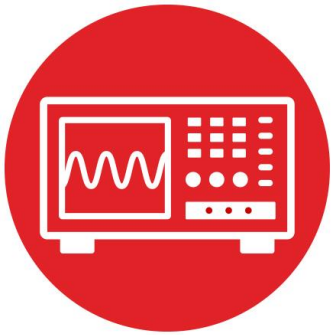

Module 18

Lab: 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

Lab: Serial Communication

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.

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