UTS

University of Technology, Sydney

Faculty of Engineering and Information Technology

Subject:	**48434 Embedded Software**
Assessment Number:	**1**
Assessment Title:	**Lab 1 – ModCon Serial Communications**
Tutorial Group:	

Students Name(s) and Number(s)

Student Number	Family Name	First Name

Declaration of Originality:

The work contained in this assignment, other than that specifically attributed to another source, is that of the author(s). It is recognised that, should this declaration be found to be false, disciplinary action could be taken and the assignments of all students involved will be given zero marks. In the statement below, I have indicated the extent to which I have collaborated with other students, whom I have named.

Statement of Collaboration:

Signature(s)

Marks

Opening comments	/0.5
Naming conventions	/0.5
Function descriptions	/0.5
Code structure	/0.5
SCI / packet functions	/2
FIFO implementation	/2
Protocol implementation	/2
TOTAL	/8

Office use only ☺

key

Assessment Submission Receipt

Assessment Title:	**Lab 1 – ModCon Serial Communications**	**Mark**
Student’s Name:		
Date Submitted:		
Tutor Signature:		Office use only ☺

Lab 1 – ModCon Serial Communications

Port access. SCI initialization. Polling. Circular buffer. Packet decoding.

Introduction

The Serial Communication Interface (SCI) is a simple yet extremely important peripheral of the MC9S12 microcontroller. On the ModCon board it is connected to a serial-to-USB chip that enables the board to communicate to a PC running Windows 7. The ModCon board is to implement the serial communication protocol as outlined in the separate document entitled "ModCon Protocol".

Objectives

1. To set up a serial communication interface on a microcontroller.
2. To implement a circular buffer in the C language.
3. To decode and respond to packets according to a defined protocol.

Equipment

- 1 ModCon microcontroller board – UTS
- 1 MC9S12 USB programmer / debugger – PEMicro
- 2 USB cables
- 1 5V DC power supply, 1 12V DC power supply, PC connector
- Freescale CodeWarrior IDE for the MC9S12

Safety

This is a Category A laboratory experiment. Please adhere to the Category A safety guidelines (issued separately).

Cat. A lab

Software Overview

The serial communication between the ModCon board and the PC uses 5-byte packets. The task of the ModCon software is to receive and send packets of information asynchronously.

This is a classic producer-consumer problem. One solution to the asynchronous nature of the communication is to implement a first-in first-out (FIFO) buffer between the producer and consumer. This is shown diagrammatically below:

Data flow graph showing two FIFOs that buffer data between producers and consumers

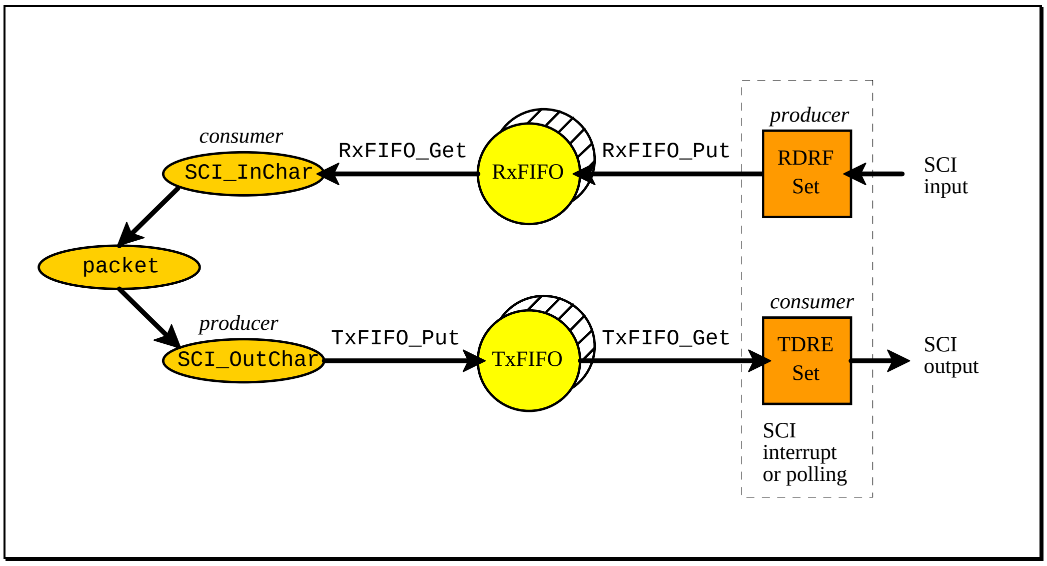


Figure L1.1

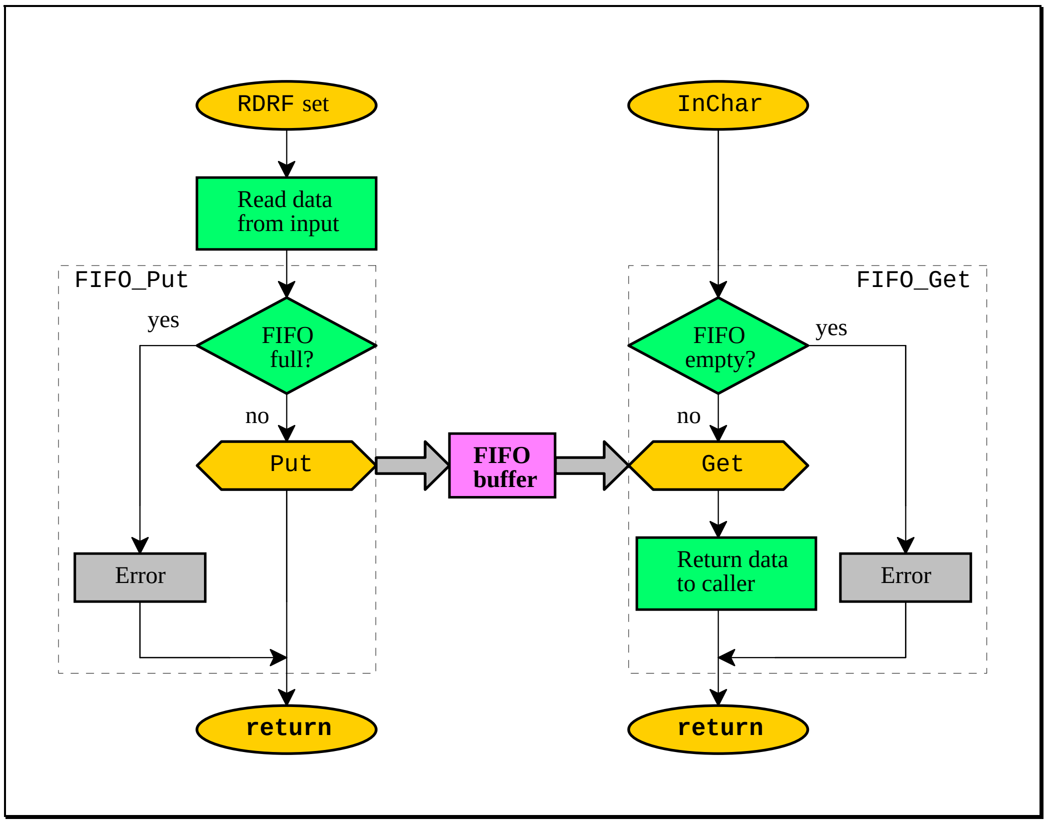
Since interrupts are not used in this particular lab, we need to simulate the asynchronous nature of reception and transmission of bytes by calling a function regularly in the main loop of the program that *polls* the status of the serial port, and calls `RxFIFO_Put` or `TxFIFO_Get` as required.

Receiving Data

When the packet module wishes to input, it calls `SCI_InChar`, which will get data from the `RxFIFO`.

The incoming serial data will set the Receive Data Register Full (`RDRF`) flag in the Serial Communication Interface Status Register 1 (`SCISR1`) register, requesting an interrupt. In the main loop, a poll of the `RDRF` flag is performed. If it is set, then the program tries to accept the data and put it in the `RxFIFO`. The `RxFIFO` buffers data between the input hardware and the main program that processes the data. If the `RxFIFO` becomes full, then data will be lost. This is illustrated below:

In this lab, the receive interrupt service routine is replaced by a polling operation



A FIFO queue can be used to pass data between an input device and a main thread

Figure L1.2

FIFO full errors will always occur if the average input rate (number of bytes arriving per second from the input hardware) exceeds the average processing rate (number of bytes processed per second by the main program). In this situation, either the output rate must be increased (by using a faster computer or by writing a better software processing algorithm), or the input rate must be decreased (by slowing down the arrival rate of data). The second way the RxFIFO could become full is if there is a temporary increase in the arrival rate or a temporary decrease in the processing rate. For this situation, the full errors could be eliminated by increasing the size of the RxFIFO.

It is inefficient, but not catastrophic, for the main program to wait on an empty RxFIFO in some cases. Efficiency can be improved for the buffered input problem by performing other tasks while waiting for data.

Sending Data

When the packet module wishes to output, it calls SCI_OutChar, which will put the data in the TxFIFO and arm the output device.

In this lab, the transmit interrupt service routine is replaced by a polling operation

The setting of the Transmit Data Register Empty (TDRE) flag by the SCI hardware signals that the output shift register is idle and ready to output more data. In the main loop, a poll of the TDRE flag is performed. If it is set, then the program tries to retrieve the data in the TxFIFO. If the TxFIFO becomes empty, then no data will be sent out the serial port. This is illustrated below:

A FIFO queue can be used to pass data between a main thread and an output device

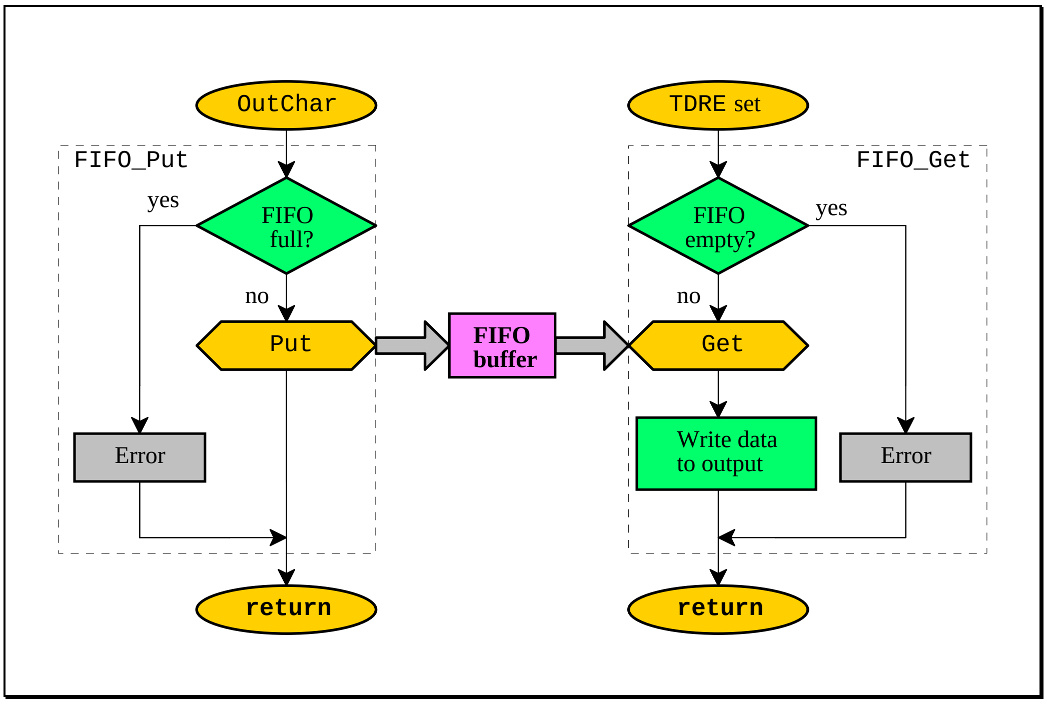


Figure L1.3

It is inefficient, but not catastrophic, for the main program to wait on a full `TxFIFO`. Efficiency can be improved for the buffered output problem by increasing the `TxFIFO` size.

Software Modules

Various modules should be written to support the serial communication protocol.

Software module dependency

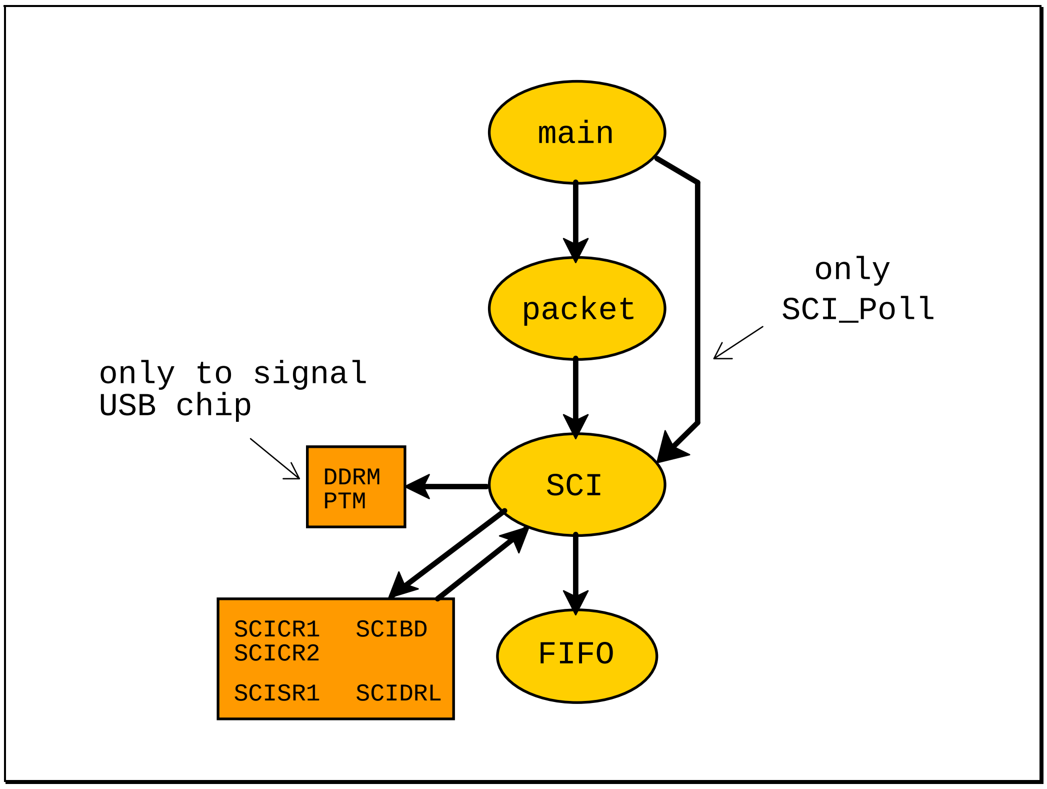


Figure L1.4

L1.6

Software Requirements

1. The baud rate is to be 38400 baud.
2. The oscillator clock is 16 MHz, and by default the bus clock (E_CLK) halves this, so E_CLK = 8 MHz (needed for baud rate divisor calculations).
3. The software must be able to handle at least 40 packets in the receive buffer and at least 40 packets in the transmit buffer (each packet is 5 bytes).
4. A polling operation can be used to check the status of the serial port in place of an ISR.
5. The commands of the ModCon serial protocol to be implemented (with packet acknowledgement) are:

ModCon to PC	**PC to ModCon**
0x04 ModCon startup	0x04 Special – Get startup values
0x09 Special – ModCon version	0x09 Special – Get version
0x0B ModCon number	0x0B ModCon number (get & set)

6. In response to reception of a “0x04 Special – Get startup values” packet from the PC, the ModCon should transmit three packets:
   - a “0x04 ModCon startup” packet
   - a “0x09 Special – ModCon version” packet
   - a “0x0B ModCon number” packet
7. Upon power up, the ModCon should send the same 3 packets as above.
8. Use the last 4 digits of your student number to initialise the ModCon number. Note that it may be changed via the “0x0B ModCon number” packet at a later time.
9. The “0x09 Special – ModCon Version” should be V1.0.
10. TortoiseSVN must be used for version control.

Marking

The software should be ready for marking on the date specified in the Timetable in the Learning Guide.

Software marking will be carried out in the laboratory, in the format of an oral exam.

Marking criteria are on the front page. Also refer to the document "Software Style Guide".

L1.8

Software Specification

The following header files are suggested, but not mandatory.

SCI.h

```
// ----------------------------------------
// Filename: SCI.h
// Description: Routines to implement the
//   transmission and reception of a byte
//   via the serial communication interface
// Author: (your name)
// Date: (the date)

// ----------------------------------------
// SCI_Setup
//
// Sets up the Serial Communication Interface
// Input:
//   baudRate is the baud rate in bits/sec
//   busClk is the bus clock rate in Hz
// Output:
//   none
// Conditions:
//   none

void SCI_Setup(const UINT32 baudRate, const UINT32 busClk)

// ----------------------------------------
// SCI_InChar
//
// Get a character from the receive FIFO if it is not empty
// Input:
//   dataPtr is a pointer to memory to store the retrieved byte
// Output:
//   TRUE if the receive FIFO returned a character
// Conditions:
//   Assumes the receive FIFO has been initialized

BOOL SCI_InChar(UINT8 * const dataPtr);

// ----------------------------------------
// SCI_OutChar
//
// Put a byte in the transmit FIFO if it is not full
// Input:
//   data is a byte to be placed in the transmit FIFO
// Output:
//   TRUE if the data was placed in the transmit FIFO
// Conditions:
//   Assumes transmit FIFO has been initialized

BOOL SCI_OutChar(const UINT8 data);
```

FIFO.h

```
// ----------------------------------------
// Filename: FIFO.h
// Description: Routines to implement a FIFO buffer
// Author: (your name)
// Date: (the date)

// Number of bytes in a FIFO
#define FIFO_SIZE 256

// ----------------------------------------
// FIFO structure
typedef struct
{
  UINT16 Start, End;
  UINT16 volatile NbBytes;
  UINT8 Buffer[FIFO_SIZE];
} TFIFO;

// ----------------------------------------
// FIFO_Init
//
// Initialize the FIFO
// Input:
//   FIFO is a pointer to a FIFO struct to initialize
// Output:
//   none
// Conditions:
//  none

void FIFO_Init(TFIFO * const FIFO);

// ----------------------------------------
// FIFO_Put
//
// Enter one character into the FIFO
// Input:
//   FIFO is a  pointer to a FIFO struct where data is to be stored
//   data is a byte of data to store in the FIFO buffer
// Output:
//   TRUE if data is properly saved
// Conditions:
//   none

BOOL FIFO_Put(TFIFO * const FIFO, const UINT8 data);

// ----------------------------------------
// FIFO_Get
//
// Remove one character from the FIFO
// Input: pointer to a FIFO struct,
//   FIFO is a  pointer to a FIFO struct with data to be retrieved
//   dataPtr is a pointer to a memory location to place the
//     retrieved byte
// Output:
//   TRUE if the operation was successful and the data is valid
// Conditions:
//   none

BOOL FIFO_Get(TFIFO * const FIFO, UINT8 * const dataPtr);
```

L1.10

Packet.h

```c
// ----------------------------------------
// Filename: Packet.h
// Description: Routines to implement packet encoding and decoding
// Author: (your name)
// Date: (the date)

// Packet structure
extern UINT8 Packet_Command, Packet_Parameter1,
  Packet_Parameter2, Packet_Parameter3;

// ----------------------------------------
// Packet_Setup
//
// Initializes the packets by calling the
// initialization routines of the supporting
// software modules
// Input: baudRate is the baud rate in bits/sec
//   busClk is the bus clock rate in Hz
// Output: none
// Conditions: none
//
BOOL Packet_Setup(const UINT32 baudRate, const UINT32 busClk);

// ----------------------------------------
// Packet_Get
//
// Attempts to get a packet from the received data
// Input: none
// Output: TRUE if a valid packet was received
// Conditions: none
//
BOOL Packet_Get(void);

// ----------------------------------------
// Packet_Put
//
// Builds a packet and places it in the transmit FIFO buffer
// Input: none
// Output: TRUE if a valid packet was sent
// Conditions: none
BOOL Packet_Put(const UINT8 command, const UINT8 parameter1,
  const UINT8 parameter2, const UINT8 parameter3);
```