METU NORTHERN CYPRUS CAMPUS

Introduction to Microprocessors | Embedded Systems Development

EEE 347 | CNG 336
Module 3

SPRING 22-23

LAB MODULE #3: HIGH-LEVEL PROGRAMMING of an EMBEDDED SYSTEM with TIMERs, INTERRUPTS and SERIAL I/O INTERFACEs

3.1 OBJECTIVE

The third laboratory module targets integration of timers, interrupts and serial I/O interfaces to the smart data logger embedded system for farming applications. These features will not only get you one step further in the design of your comprehensive embedded system, but will also improve the cost-effectiveness, efficiency and reliability of the system. You will use Module 3 as an opportunity to explore high-level programming of MCUs in developing critical subsystems. It is very important that you start Module 3 with Module 2 system in fully functional state. Any missing pieces from Module 2 should be completed first. The following four new features will be the primary additions in this module to the embedded system you developed in Module 2:

i. The user communication interface that was emulated using switches and LEDs in Module 2 will be replaced by a Bluetooth interface, accessible through serial USART port, for providing wireless short-range connection to personal mobile devices, such as cell phones;
ii. the data communication interface, which was also emulated using switches and LEDs in Module 2, will be replaced by an RF interface, accessible through another serial USART port, for providing wireless long-range connection to remote sensor nodes,
iii. efficiency of the system will be improved by converting polling-based operations in Module 2 to interrupt-driven operations and taking advantage of MCU power management modes,
iv. reliability of the system will be improved through the addition of an EEPROM-configurable ‘watchdog’ timer to track and try to minimize unexpected down times at the smart datalogger and at the remote sensor nodes.

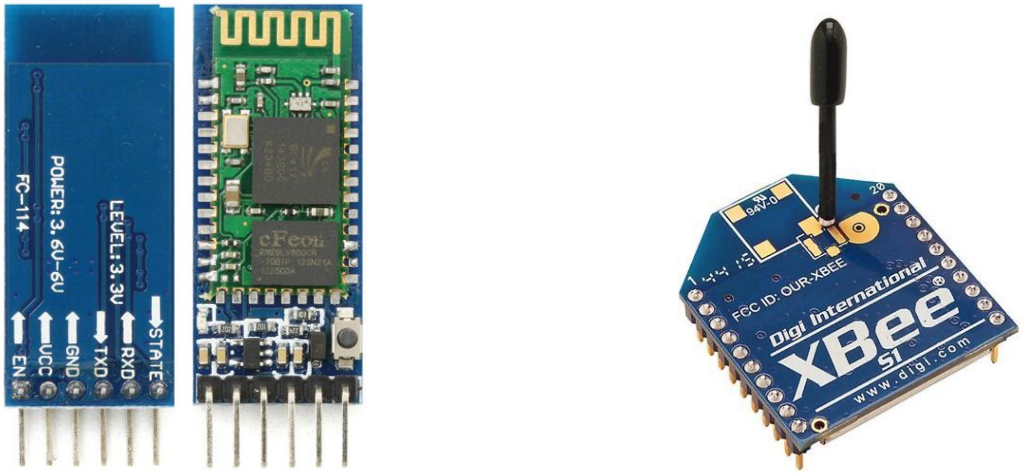
You will follow a modular approach as before, taking advantage of subroutines for organization. You will then debug and simulate the code first on Microchip Studio, followed by embedded system on Proteus, paying attention to different operation modes and corner cases. Design and simulation results should be well documented in your submitted report before your scheduled laboratory session. You will then be prepared to quickly and effectively demonstrate your development and verification process to the lab instructor by sharing the Microchip Studio and Proteus sessions directly from your computer, and answering any questions. You need to be ready to create various scenarios by entering different operational system modes upon request. Do not worry if some details of your design are not fully ironed out. You will continue improving your % functionality as well as coding friendliness and efficiency in Module 4 as you develop final features for your project.

3.2 BACKGROUND

3.2.1 Wireless user and data communications

Personal mobile devices such as smart phones provide practical interfaces to users that are always available since such devices are carried typically on the user at all times. Many such electronic computing platforms nowadays implement one version of wireless Bluetooth technology for quick practical short-range point-to-point communications with another device with the same capability. You will incorporate an HC-05 Bluetooth transceiver (transmitter + receiver), depicted in Figure 3.1(a), to facilitate user interface with up to ~20 meters of range, using one of the USART subsystems within ATmega128.

Long-distance wireless data communications are typically supported through radio-frequency (RF) transceivers through embedded system components such as Zigbee, Xbee, and other. The second onboard USART subsystem will be used to integrate an Xbee Series 1 (S1) transceiver, shown in Figure 3.1(b), into your system for wireless communications with remote sensors up to 1.6 km. An embedded system designer has to be always aware of the tradeoff between range and power dissipation when integrating I/O modules for communications. Ultimate range requirements of a given application needs to be carefully considered. While HC-05 Bluetooth module consumes about 40 mW during normal operation, Xbee S1 may consume up to 700 mW in order to achieve longer range for transmission. Actual specifications depend on the version and features of the modules.



xy dimensions: 27mm **x** 13mm; weight: 3.5g
(a)

xy dimensions: 27mm **x** 24mm ; weight: 4g
(b)

Figure 3.1. (a) HC-05 Bluetooth transceiver (left), and (b) Xbee Series-1 RF transceiver (right) modules

3.2.2 Interrupt-driven operation with power management modes

You have observed in Module 2 that running multiple tasks in a system can be achieved by serializing all through separate polling loops, where polling implies wasting time by running instructions that will keep checking if a particular event, e. g. a pushed button, has happened. Polling prevents one from completing other tasks while waiting for these events. It also prevents a microcontroller (or microprocessor) from entering a lower power state while waiting for a meaningful event to happen. It is therefore inherently inefficient in terms of getting multiple tasks completed fast and with as little power dissipation as possible. Interrupts used in conjunction with MCU low-power states, therefore, are essential features of contemporary multi-tasked systems.

Microcontrollers and microprocessors may have different power management states with fewer or more subsystems kept active (or alive), while the other subsystems are turned off by shutting down their clocks or turning off a local power supply. In microprocessors, for example, DEEP-SLEEP mode is a lower-power state than SLEEP mode. ATmega128 has six power management / sleep modes entered using the SLEEP instruction. The modes are configured using Sleep Enable (SE) bit along with SM[2:0] bits in MCUCR (MCU Control Register): i. Idle, ADC Noise Reduction, Power-down, Power-save, Standby, Extended Standby. Standby and Extended Standby modes are only available when external crystals or resonators are used.

You will replace polling loops from Module 2 with interrupts in Module 3, and will put ATmega128 in IDLE mode when waiting for meaningful events to happen. Adding power management C library to your code will help coding for power management, although you do not need to necessarily use special functions when you can configure any register through direct assignments: `avr/sleep.h`. For example, SM bits can be configured by **set_sleep_mode()** function with a passed argument to determine the sleep mode e.g. IDLE in this case. **sleep_enable()** function turns on the SE bit in MCUCR to enable low power state. SE bit should only be set immediately before turning on the interrupts and putting CPU into low power state (through **sleep_cpu()** function) while waiting for the interrupts. When the CPU receives an interrupt and wakes up to handle it, execution resumes from the final SLEEP instruction. Low power mode should be disabled through **sleep_disable()** function, while interrupt is handled. Once all handling is completed, the CPU can be put to sleep again to wait for the next interrupt. A typical sequence in a subroutine, which will allow you to put the MCU to sleep and wait for interrupts, may look like this:

```
…
sleep_enable();     // arm sleep mode
sei();              // global interrupt enable
sleep_cpu();        // put CPU to sleep
sleep_disable();    // disable sleep once an interrupt wakes CPU up
```

3.2.3 Watchdog timers

Watchdog timers count from an initialized value to 0 using an internal oscillator and then take the processor through a reset sequence to start fresh again. In normal operation, an enabled watchdog timer should not be allowed to expire. Watchdog timers allow processors to keep track of failures that cause unexpected operation. For example, a watchdog timer can be initialized to expire in 4 ms at the beginning of a task that is expected to take no more than 1 ms. If an unexpected problem happens during the execution of the task that causes it to hang without completion, then eventually watchdog timer will expire and start a reset sequence. Watchdog timers should be enabled only when necessary because they typically cause additional power dissipation even in low-power states. For more details on Watchdog timers, please refer to AVR ATmega128 datasheet, page 54.

User interface will be enriched in Module 3 to receive input from the user about ATmega128 Watchdog timer enabling, and configuring this information to EEPROM for availability after power-cycling. At the same time a second watchdog timer will also be programmed using one of the existing 16-bit timer/counters in order to send a Reset-request to the remote sensor node, if no new data packet has been received within a certain time interval. This interval will also be configured by writing a 16-bit value to the EEPROM based on user input.

3.2.4 Virtual Terminal tool in Proteus

Proteus offers a Virtual Terminal tool under Virtual Instruments Mode, to emulate simple serial links with different configurations. A full duplex configuration is depicted in Figure 3.2. Virtual Terminal accepts normal logic levels, so that there is no need to introduce interface circuits between transmitters and receivers. To configure the serial communication parameters in the Virtual Terminal, right click on the Virtual Terminal symbol and choose properties. Figure 3.3 shows the properties dialog of the Virtual Terminal. Be sure to have a consistent configuration at both ends of any serial link in order to avoid communication problems.

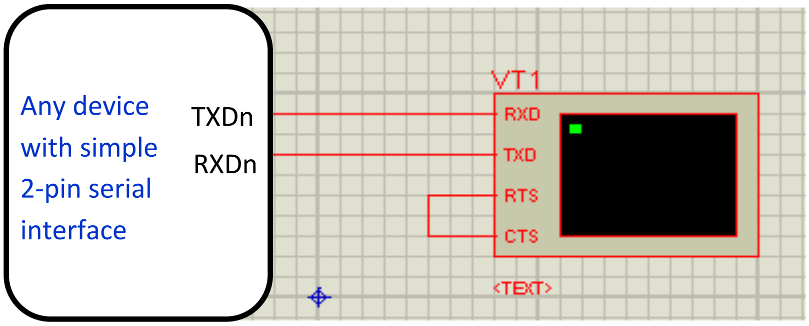
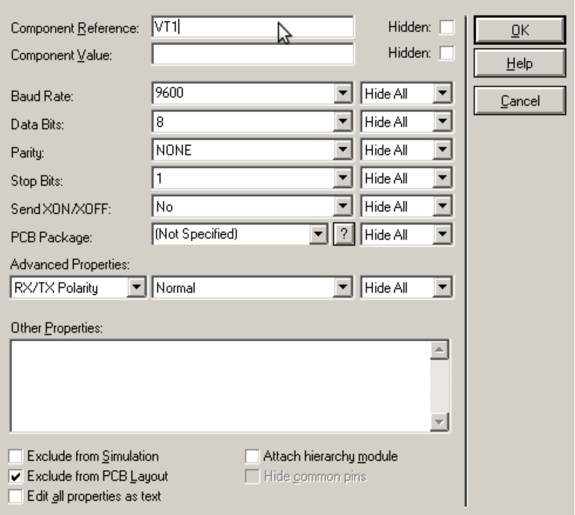


Figure 3.2. Virtual Terminal in full duplex mode



Component Reference:	VT1		Hidden: □
Component Value:			Hidden: □
Baud Rate:	9600	Hide All	
Data Bits:	8	Hide All	
Parity:	NONE	Hide All	
Stop Bits:	1	Hide All	
Send XON/XOFF:	No	Hide All	
PCB Package:	(Not Specified)	Hide All	
Advanced Properties:			
RX/TX Polarity	Normal	Hide All	

Other Properties:

□ Exclude from Simulation
☑ Exclude from PCB Layout
□ Edit all properties as text
□ Attach hierarchy module
□ Hide common pins

Figure 3.3. Virtual Terminal properties dialog box

3.2.5 Integrating Virtual Terminal, Bluetooth and Xbee modules into Proteus Project

Virtual Terminal (VT), readily available in Proteus libraries, will be used to emulate interfaces at the User end and Remote Sensor end. Therefore, you will need to establish various serial communication connections correctly in your system as described in this section. **It cannot be emphasized enough that MCU clock frequency (you may use 8 MHz internal clock for this lab, no external clock necessary), and Baud Rate configuration of the USART interfaces should all be consistent**. It has been reported statistically that 99% of problems in Proteus simulations involving serial communications occur due to a mismatch in one of these settings. If you do not have it right, you will get corrupted data across the communication.

Between a VT module and BLUETOOTH module (user side):

Wiring is depicted in the top left portion of Figure 3.6 in the next section. RX pin of the VT connects to TX pin of the BLUETOOTH module and vice versa. Both VT and BLUETOOTH modules should be operating with the same Baud Rate, and similar serial communication setting. If you double-click on the BLUETOOTH module, the correct configuration that is consistent with VT settings in Section 3.2.4 will be as follows: Physical port number, set to COM3 in this example, is not critical for connection between VT and BLUETOOTH. However, as described in the next paragraph, the port number you pick here needs to be unique to facilitate the rest of the setup. Since the VT in this subsystem is emulating a user interface, VT will be utilized in regular human readable character display mode. Therefore, when you right-click on the VT window, "Echo typed character" should be selected, but "Hex display mode" should not be selected.

Between two BLUETOOTH Modules (wireless communications):

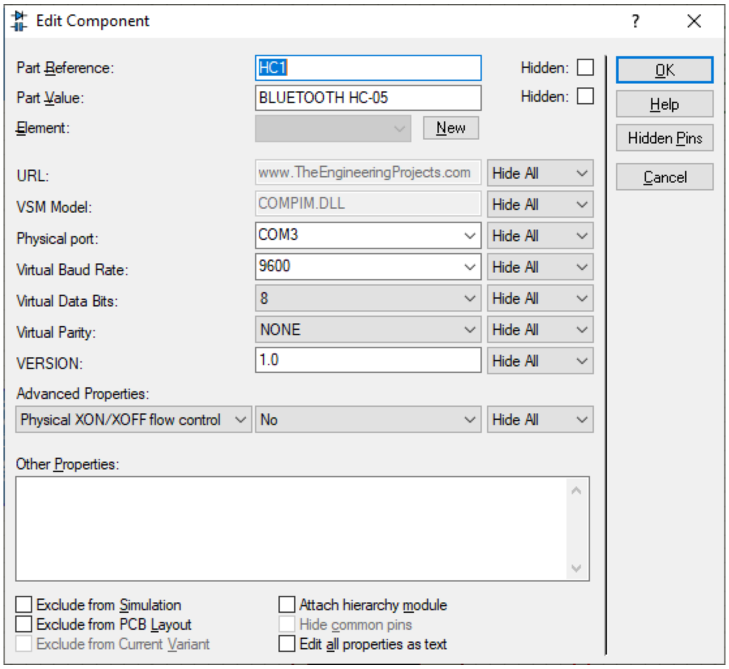
The wireless communication between two BLUETOOTH modules will be emulated in your simulations through pairing between the COM ports of the modules. You will install the provided Virtual Serial Ports Emulator (VSPE) in order to establish this pairing. After instantiating the second BLUETOOTH module in your Proteus project, configure it similar to Figure 3.4 in the next page, except use a different unique COM number, e.g. COM2 in this example. Then launch VSPE, declining ordering of any licences by clicking 'No'. Hit the fifth icon from left in the upper menu to enter a new device (Figure 3.5 left). Then select "Pair" option for Device type (Figure 3.5 center), click "Next", and set the virtual serial ports in the next screen to the PORT numbers you used for two BLUETOOTH modules in your Proteus project (Figure 3.5 right). Then hit "Finish", but do not close the VSPE tool as long as you are simulating wireless communications in your Proteus project. You may verify successful process up to this point by connecting another VT to the second BLUETOOTH module, and check communications between the two by typing a text to the VT connected to one module, and watching the text appear on the terminal of the second module. If this does not happen, your BAUD rates, COM settings, or pairing configuration may have failed.

Between a BLUETOOTH module and ATmega128 (smart logger side):

As shown on the bottom left side of Figure 3.6 in the next section, second BLUETOOTH module enables wireless communication with the user side for the MCU. USART0 RX pin should be connected to second BLUETOOTH module's TX pin and vice versa.

Between a VT module and XBEE S1 module (remote sensor side):

Wiring is depicted in the top right portion of Figure 3.6 in the next section. As observed, Proteus XBEE module connectivity requirement is inverted, compared to the Proteus BLUETOOTH module. RX pin of the VT connects to RX pin of the XBEE module and vice versa. Otherwise, Baud Rate settings should be same as before, but each XBEE module should have its own unique COM number that is different from the COM numbers you used for BLUETOOTH modules. The properties dialog box picture has not been provided here because it is the same as the one for BLUETOOTH. Since the VT in this subsystem is emulating a sensor node (machine) interface, VT will be utilized in hexadecimal display mode. Therefore, when you right-click on the VT window, both "Echo typed character" and "Hex display mode" should be selected.



Part Reference: HC1 Hidden: □
Part Value: BLUETOOTH HC-05 Hidden: □
Element:

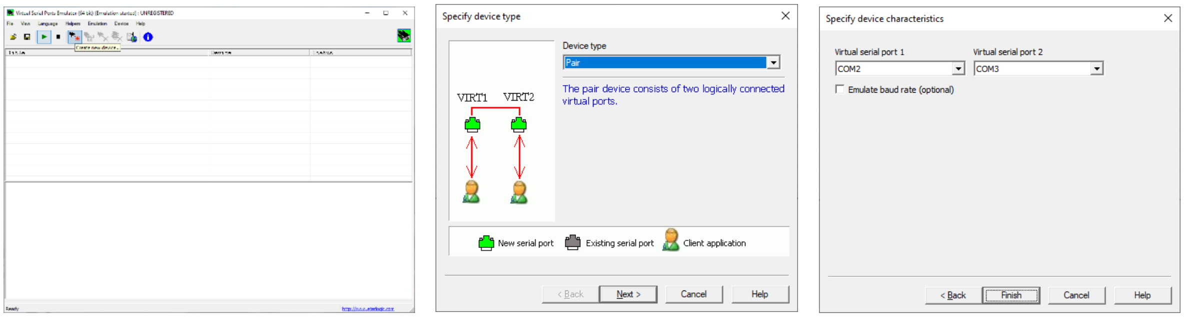
URL: www.TheEngineeringProjects.com Hide All
VSM Model: COMPIM.DLL Hide All
Physical port: COM3 Hide All
Virtual Baud Rate: 9600 Hide All
Virtual Data Bits: 8 Hide All
Virtual Parity: NONE Hide All
VERSION: 1.0 Hide All

Advanced Properties:
Physical XON/XOFF flow control No Hide All

Other Properties:

□ Exclude from Simulation □ Attach hierarchy module
□ Exclude from PCB Layout □ Hide common pins
□ Exclude from Current Variant □ Edit all properties as text

Figure 3.4. BLUETOOTH HC-05 module properties dialog box



Virtual Serial Ports Emulator (64 bit) (Emulation started) : UNREGISTERED
File View Language Helpers Emulation Device Help
Create new device...
Ready
http://www.eterlogic.com
Specify device type
VIRT1 VIRT2
New serial port
Existing serial port
Client application
< Back
Next >
Cancel
Help
Specify device characteristics
□ Emulate baud rate (optional)
< Back
Finish
Cancel
Help

Device type: Pair

The pair device consists of two logically connected virtual ports.

Virtual serial port 1: COM2

Virtual serial port 2: COM3

Figure 3.5. Virtual Serial Ports Emulator (VSPE) dialog boxes

Between two XBEE Modules (wireless communications):

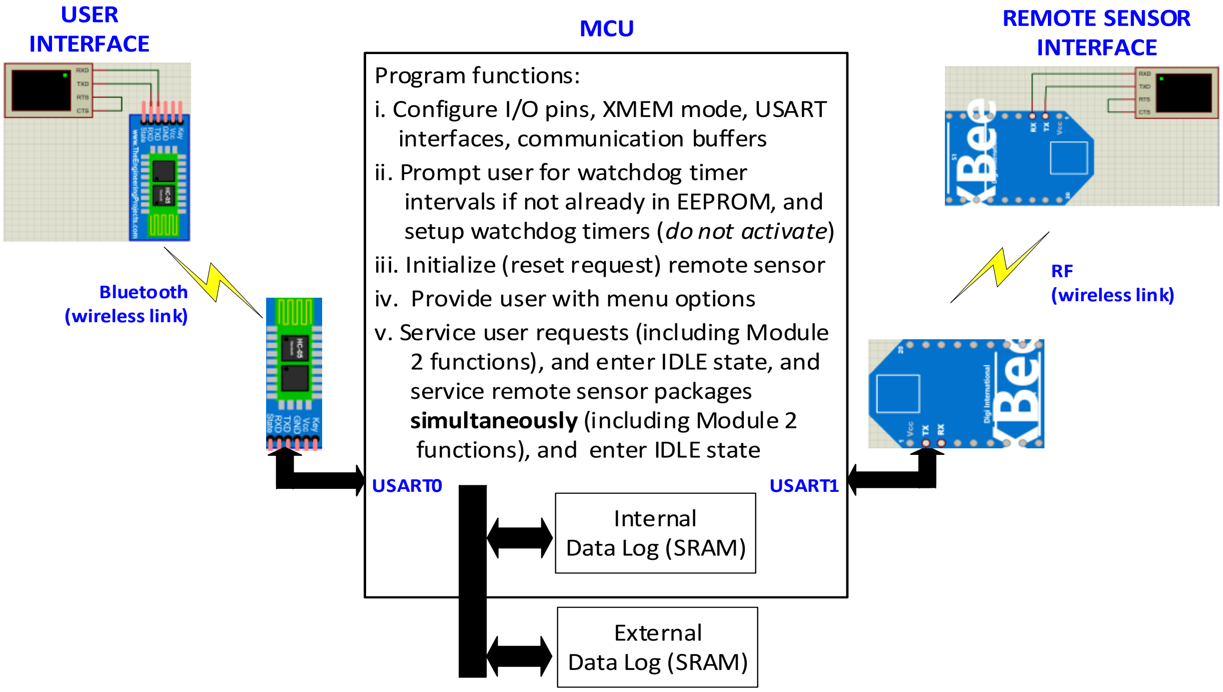
After instantiating the second XBEE module, go through the same procedure you did for BLUETOOTH modules to ensure pairing between the unique COM ports dedicated to the wireless XBEE modules. For example, if you used COM2 and COM3 for BLUETOOTH modules, you may use and pair COM4 and COM5 for XBEE modules. You may test the communication between the modules in a similar manner as before using VTs attached to both modules.

Between an XBEE module and ATmega128 (smart logger side):

As shown on the bottom right side of Figure 3.6 in the next section, second XBEE module enables wireless communication between the MCU and the remote sensor. XBEE connectivity is non-standard in Proteus: USART1 RX pin should be connected to second XBEE module's RX pin, and TX pin to module's TX pin.

3.3 PROBLEM DESCRIPTION

The *smart data logger system* will be revised to be closer to its final form in this phase of the project. The user interface will be converted to wireless Bluetooth and data interface will be converted to Xbee wireless radio communication through serial exchange of commands/data between ATmega128 and these modules through on-board USART subsystems, as depicted in Figure 3.6. As shown in the same figure and described in detail in the Background section, the other ends of both communication links are going to be modeled using Proteus Virtual Terminal that emulates user touch-screen I/O at the user interface (left), and MCU I/O at the remote sensor interface (right). The core of the system functions associated with user options and command/data packets exchanged with the remote sensor are the same as in Module 2. However, USART write/read protocols will replace the previous transmit/receive protocols devised in Module 2 to work with temporary emulation of the interfaces using LEDs and switches, in order to debug the core functionality.



Program functions:

i. Configure I/O pins, XMEM mode, USART interfaces, communication buffers

ii. Prompt user for watchdog timer intervals if not already in EEPROM, and setup watchdog timers (*do not activate*)

iii. Initialize (reset request) remote sensor

iv. Provide user with menu options

v. Service user requests (including Module 2 functions), and enter IDLE state, and service remote sensor packages **simultaneously** (including Module 2 functions), and enter IDLE state

Figure 3.6. Main functions and interfaces of the improved smart data logger.

3.4 DESIGN AND REPORTING

3.4.1 Preliminary Questions

Answer the following questions to enhance your understanding of the described system.

a) You will need to think carefully about how you want to setup different types of interrupts in your system. What are the different types of interrupts in this system and which interrupts should be allowed to occur at the same time?

b) Study ATmega128 datasheet and explain how the watchdog timer feature is enabled and how it is programmed to support different watchdog delays. Provide a sequence of instructions for Watchdog timer configuration for different delays and enabling.

c) Study the datasheets for HC-05 Bluetooth and Xbee Radio.
   i. When do each of these components dissipate maximum power? Is it when the module is receiving data, transmitting data, or neither receiving nor transmitting?
   ii. What do you think is the reason behind the specified behavior in (i).

d) Study ATmega128 datasheet and carefully explain the differences in six different power management / sleep modes. Which one do you think represents the lowest power mode? Clarify your assumptions.

e) After studing Figure 3.4 above and Section 3.4.2 below, draw an algorithmic flowchart, similar to the one you were provided in Module 2, to outline how your overall code will work. Consider the design guidelines and subroutines described in Section 3.4.2 in sketching your flowchart.

3.4.2 Design

The following approach in the implementation of high-level C functions is recommended to divide and conquer the design. Because the task at hand is relatively complex (which mimics real-life embedded system design), segments of code are provided to help you make progress. The goal in the exercise will be to fit all text messages, and global variables in the data memory area between 0x100 – 0x500. In any embedded system, one needs to be aware of the available memory resources. It is recommended that you maintain all of your constants and messags that will go to the user during operation at the beginning of your code using ***#define*** statements. This means your data memory for logging data from remote sensor will start from 0x500, and will extend to external memory by skipping over a reserved stack space of 50 addresses at the end of the data memory, similar to Module 2.

a) Libraries and global declarations section:

Here are C libraries that you may find useful, although you certainly may not need all, depending on which standard C functions you use in coding especially for handling characters and strings:

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Before you get started determine how you will detect the end of a user input from USART0. In this example, we use a period ('.') character. For example, when you transmit the following menu on USART0 to user interface, you will expect a user to type in only two characters as **'2' followed by '.'** to pick displaying the last entered byte to memory:

```
#define USER_MENU "\rEnter choice (and period): 1-Mem Dump 2-Last Entry 3-Restart  \0"
```

It is recommended to maintain different global serial buffers for data flowing in and out of the MCU. Your core algorithms should only deal with these data buffers, and the interface between these buffers and USART interfaces should be done in separate routines. In your case, the message you will transmit to user may take up the most buffer space, while the transmission to remote sensor side or received messages from user menu selections or remote sensor packets will only be one or two bytes at most. You may save static data memory space, by defining your buffers accordingly. For example:

```
    // Maximum buffer sizes for transmission:
    #define USER_TR_BUFFER_SIZE 128
    #define SENSOR_TR_BUFFER_SIZE 5
    // USART0 communication buffers and indexes
    unsigned char user_tr_buffer[USER_TR_BUFFER_SIZE] = "";
    unsigned char user_tr_index = 0;   // keeps track of character index in buffer
    unsigned char new_user_read_char;  // reception one character at a time
    // USART1 communication buffers and indexes
    unsigned char sensor_tr_buffer[SENSOR_TR_BUFFER_SIZE] = "";
    unsigned char sensor_tr_index = 0;
    unsigned char new_sensor_read_char;
```

It may also be useful to declare global pointers to keep track of data logging entries and their reporting based on user menu selections:

```
    // Data memory pointers
    unsigned char *x = IntMemStrt;  // points to the next data memory entry
    unsigned char *z = IntMemStrt;  // used in memory dumps
    unsigned char *y = ExtMemStrt;  // can be used in memory dumps, if one wishes
```

b) Managing the complexity through divide-and-conquer

Think about the fundamental tasks that need to be done by your code first. Here is an example sequence:

i. Setup all I/O pins including the ones you would like to use for debug, configure MCU for external memory and USART communications, and initialize any data buffers for serial communication

*ii. Ask user input to configure Watchdog Timer delays for the smart datalogger and remote sensor, unless EEPROM already has these configurations from a previous session (**this can be done using polling or interrupts**)*

*iii. Initialize remote sensor by sending a reset packet – you may want to let user know about it (**this part can be done using polling or interrupts**)*

iv. Display user menu options, for example, as shown in section 3.4.2(a) and ask for input.

*v. Go into an infinite loop, **enabling interrupts**, and waiting for inputs from user interface and remote sensor interface (packet_in) at the same time; keeping track of interrupts generated asynchronously by the received communication from both USART 0 and 1, and responding to user menu selections on one side by displaying information to user screen, and responding to packet_in's on the other interface by logging data and generating appropriate packet_out's to send to remote sensor (consistently with Module 2 work).*

c) Next level of detail: Modular and readable design through functions (subroutines)

i. Sample interrupt handlers (again what you choose to do in a handler may be a bit different):

```
// Sample Interrupt handler for USART0 (User side) when a TX is done
ISR(USART0_TX_vect) {
      if (user_tr_index == 0){   // transmit buffer is emptied
             UserTrBufferInit(); // reinitialize the buffer for next time
             UCSR0B &= ~((1 << TXEN0) | (1 << TXCIE0));  // disable
interrupt
      }
// Sample Interrupt handler for USART0 (User side) when an RX is done
ISR(USART0_RX_vect) {
             while (! (UCSR0A & FULL_BYTE_RECEIVED)){}; // Double checking
flag
             new_user_read_char = UDR0;
}
```

ii. Example of saving energy while transmitting everything in a transmit buffer by sleeping, instead of polling:

```
// Empty the user transmit buffer using interrupt, saving precious energy
void UserBufferOut(void) {
             unsigned char i = 0;
             // Enable TX to user (USART0 TXen and TX complete interrupt)
             UCSR0B |= (1 << TXEN0) | (1 << TXCIE0);
             while (user_tr_index > 0) {
                    user_tr_index--;
                    UDR0 = user_tr_buffer[i++];
                    Sleep_and_Wait();
             }

}
```

iii. As part of Task 1 (i) in the previous section, you may want to initialize your defined buffers so they start out with zero index and no content. Send a message to user on USART0 to inform when MCU intializations are completed (may use polling or interrupts).

iv. As part of Task 2 (ii) in the previous section, check the first 2-byte entry in EEPROM. If it is 0xFFFF (EEPROM content is set ‘high’ when not programmed), then write a function ***ConfigMasterWD*** with following content: ***ConfigMasterWD*** sends (and prints) a text message to User screen to ask for MCU Watchdog timer delay. You should then receive the user selection from the user interface and configure the first 2-bytes (you may also use one more Bytes, depending on your design) of EEPROM appropriately to reflect the user selection. Check the next 2-byte entry in EEPROM. If it is 0xFFFF, then write a function ***ConfigSlaveWD*** with following content: ***ConfigSlaveWD*** sends (and prints) a text message to User screen to ask for Remote Sensor Watchdog timer delay, by providing the following three options in the text: 0.5s, 1s, 2s. You should then receive the user selection from the user interface and configure the second 2-bytes (you may also use one or more Bytes, depending on your design) of EEPROM appropriately to reflect the user selection. Timer 1 will be used to configure slave watchdog such that whenever slave watchdog expires without receiving a packet_in, then remote sensor is re-initialized. Here is a sample message definition for prompting user choice:

```
#define MST_WD_MENU "\rEnter MS WD Choice (& period):\rA-30ms\rB-250ms\rC-500ms  \0"
```

IMPORTANT: You will use this lab exercise to get familiar with watchdog timers and how to configure them, but you WILL NOT enable master watchdog interrupts on your MCU during your debug to avoid having to deal with frequent resets in your simulated system. Just be ready to demonstrate watchdog operation during your demo, if/when asked.

v. Remember when handling Task 3 (iii) and the following remote sensor tasks in the previous section that every Packet_Out has to be appended with a ***CRC3*** remainder, and every Packet_In has to go through a ***CRC3_check*** or ***CRC11_check***, as you have done in Lab Module 2. This means you have to translate your corresponding subroutines from assembly code to high-level C code for Module 3. Here is a sampe C function, for example, in order to accomplish Task 3, consistently with the implementations you have done in lab Module 2:

```
void Init_RemoteSensor(void) {  // A function to initialize remote sensor
             // Note how TOS becomes an 8-bit variable in Lab module 3:
             TOS = CRC3(RESET_COMMAND);
             // Send sensor an initialization packet
             MessageSensor(TOS);
             // Be nice enough to inform user about initializing the sensor
             strcpy(user_tr_buffer, SENSOR_RST_INF_MSG);
             MessageUser();
}
```

3.4.3 Verification

a) Use Microchip Studio to create Module3_MicrochipStudio project. Debug different functions incrementally to make sure the code works as intended, and take screen captures (PrtScr) to illustrate different modes. When debugging your program with interrupts in the Microchip studio debugger in step mode, you need to have the following option in the tool (or the debugger will not recognize interrupts in step mode):

Tools Menu ➔ Options ➔ Tools ➔ Tool Settings ➔ Mask interrupts while stepping ➔ False

Microchip Studio will freeze if you activate Sleep mode on the MCU and there are no generated interrupts (e. g. you did not setup the interrupts correctly). In that case you will need to go into Task

Manager on your PC and kill the Microchip Studio session and restart it. **Therefore, it is particularly important in this lab exercise to do frequent saving of your work.**

b) Use Proteus to create Module3_Proteus project. Instantiate and connect ATMEGA128 microcontroller, Xbee, HC-05 and virtual terminals as in Figure 3.4. Load your debugged object code, and run different cases, especially demonstrating each mode of operation, communication with user and with remote sensor, and correct logging of data to internal and external memory. Take screen captures (PrtScr) again for the critical cases to include in your report. If you would like to add additional LEDs to certain DEBUG pins on parallel ports to make your debugging job easier, you are welcome to do so. It is recommended that you utilize Proteus VSM debugging mode, and use stepwise simulation to watch activities by selecting visual aids under “Debug” menu, such as Watchwindow, CPU registers, Data Memory, I/O registers, etc.

3.4.4 Report

Follow the strict guidelines and format described in detail in the first lab manual to complete a concise and comprehensive report. Your report should represent your team’s work only, and should clearly document your solutions to preliminary exercises (3.4.1), your modular and organized code, MCU and full embedded system critical verification results along with Proteus schematics. Each simulation screen you include should be carefully annotated and explained in a paragraph. If there are any problematic cases that do not function correctly, these should be discussed. It may be ok not to have 100% functionality, as long as you have achieved most of the stated goals, and have a good plan to continue to debug the rest of the problems before you can demonstrate your final project at the end of the semester.

Note on Attachments:

- Bluetooth HC-05 and XBEE Series-1 RF transceiver module files should be placed in Proteus library directory before they can be used in your project as any other component. The LIBRARY directory for Proteus is typically under the following path, or in your case it could be a similar path. It should specifically be named "LIBRARY" folder, with other components in it.

  C:\ProgramData\Labcenter Electronics\Proteus 8 Professional\LIBRARY
- VSPE setup should be separately executed, and is quick to install.