ECE 375: Computer Organization and Assembly Language Programming

Lab 8 – Remotely Operated Vehicle

SECTION OVERVIEW

Complete the following objectives:

- Use your knowledge of computer organization and assembly programming to create a real system, which will serve as a proof-of-concept for a possible consumer product.
- Improve your knowledge of interrupts, and your ability to coordinate interrupts from multiple sources.
- Write two separate assembly programs (for two separate `mega128` boards) and have them interact.
- Learn how to configure and use the *Universal Synchronous/Asynchronous Receiver/Transmitter* (USART) module on the ATmega128 microcontroller.

PRELAB

To complete this prelab, you may find it useful to look at the full ATmega128 datasheet. If you consult any online sources to help answer the prelab questions, you **must** list them as references in your prelab.

1. In this lab, you will be given a set of behaviors/actions that you need to have a proof-of-concept "toy" perform. Think of a toy you know of (or look around online for a toy) that is likely implemented using a microcontroller, and **describe the behaviors it performs**. Here is an example behavior: "If you press button X on the toy, it takes action Y (or makes sound Z)".
2. For each behavior you described in the previous question, explain which microcontroller feature was likely used to implement that behavior, and give a brief code example indicating how that feature should be configured. Make your explanation **as ATmega128-specific as possible** (e.g., discuss which I/O registers would need to be configured, and if any interrupts will be used), and also mention if any additional mechanical and/or electronic devices are needed.
3. Each ATmega128 USART module has two flags used to indicate its current **transmitter** state: the Data Register Empty (UDRE) flag and Transmit Complete (TXC) flag. What is the difference between these two flags, and which one always gets set first as the transmitter runs? You will probably need to read about the *Data Transmission* process in the datasheet (including looking at any relevant USART diagrams) to answer this question.
4. Each ATmega128 USART module has one flag used to indicate its current **receiver** state (not including the error flags). **For USART1 specifically**, what is the name of this flag, and what is the interrupt vector address for the interrupt associated with this flag? This time, you will probably need to read about *Data Reception* in the datasheet to answer this question.

BACKGROUND

As previous labs have demonstrated, microcontrollers are well-suited for tasks such as: receiving user input and generating output via general-purpose I/O (GPIO) ports, performing arithmetic and logic computations, and accurately measuring intervals of time. However, microcontrollers are also frequently used for tasks that require communication with the "outside world", which for our purposes means anything that exists outside of the `mega128` board.

When using a microcontroller to interact with the outside world, it is very convenient to have a structured, *standardized* way of exchanging information, as opposed to manually implementing data transfer using the GPIO ports. For this reason, microcontrollers often implement a variety of *communication protocols*.

Communication protocols come in two varieties: *parallel* and *serial*. Parallel communication protocols are typically used for internal microcontroller communication, such as a shared data bus. Serial communication protocols are more commonly used for external interactions. Some common examples of serial protocols are *Serial Peripheral Interface* (SPI) and *Two-wire Serial Interface* (TWI). (The LCD Driver from Lab 4 uses SPI to configure and send data to the LCD.)

The ATmega128 microcontroller, in addition to providing built-in modules for both SPI and TWI, also comes with two *Universal Synchronous/Asynchronous Receiver/Transmitter* (USART) modules. USART is not a communication protocol; instead, it is a highly-configurable hardware module that can be setup to implement point-to-point serial communication (with various parity, data rate, and frame format settings). If two microcontrollers each have a USART module, then they can perform *full-duplex* serial communication as long as their respective USART modules are configured with the same settings, and as long as there is a physical interface (wired, or wireless) that links their respective TX/RX pins.

PROCEDURE PART 1 – REMOTE CONTROL

Problem Statement

Imagine that you now work for TekToy Corporation, a global subsidiary of TekBots International. You have been given the task of designing a new toy for the "remotely controlled" line of products, and you intend to build a simple robot that can be controlled using an infrared (IR) remote. You plan to use IR because it is cheaper and requires less design than an equivalent radio frequency (RF) system.

Your first step is to build a proof-of-concept version of the robot, so that you can sell the idea to management. You dig through your desk and come across a TekBot base and a couple of `mega128` boards, leftover from when you were in school. Since you are familiar with these parts, you decide to use them to build your proof-of-concept toy. One of the `mega128` boards will be used as the remote, and the other will be used as the robot.

Specifications

1. The `mega128` boards you are using for your proof-of-concept have a built-in IR transmitter and receiver, which are connected to the transmit and receive pins of one of the ATmega128's USART modules. Since you wanted to use IR anyway for this project, using a USART module to facilitate communication between the remote and robot seems like a good choice.

   You will need to configure the USART module on the robot board **and** on the remote board. Also, although the ATmega128's USART modules can communicate at rates as high as 2×10^6 bits per second (2 Mbps), the built-in IR transmitter and receiver cannot work this fast. Instead, you will use the (relatively) slow baud rate of **2400 bits per second**.

2. For your simple proof-of-concept robot, you only need to implement a few different actions: move forward, move backward, turn right, turn left, and halt. A user should be able to select from these actions using the pushbuttons on the remote. Once the robot receives an action from the remote, it must continue performing that same action until a different action is received, without needing to receive the same action repeatedly from the remote.

3. Management will insist that multiple robots can successfully run at the same time in the same room, so you need to somehow make a robot respond **only to its own remote control**. You decide to solve this problem by assigning each robot a distinct address, which the remote will transmit along with every selected action.

4. A very basic protocol starts taking shape at this point in your design process. At a high level, any action sent from a remote to a robot will be sent as a 16-bit logical "packet". At a lower/more detailed level, this 16-bit packet actually consists of two 8-bit values that will be sent back-to-back by the remote's USART module. The first 8-bit value will be a "robot address" byte, which indicates which specific robot the packet is intended for. The second 8-bit value will be an "action code" byte, which indicates which action the user wants the robot to take. You decide that 8-bit robot addresses must begin with (have an MSB of) "0", and that 8-bit action codes must begin with (have an MSB of) "1". Table 1 depicts the packet structure that you have decided to use:

Byte 1: Robot Address	Byte 2: Action Code
0 X X X X X X X	1 X X X X X X X

Table 1: Packet Structure for Remote-to-Robot Communication

5. To make your life easier, you decide to specify the action codes for each of the different actions ahead of time, before you even begin writing any code. These 8-bit action codes are shown in Table 2:

Robot Action	Action Code
Move Forward	0b10110000
Move Backward	0b10000000
Turn Right	0b10100000
Turn Left	0b10010000
Halt	0b11001000
Future Use	0b11111000

Table 2: Action Codes

6. You would also like to have some minimal intelligence on the robot (in other words, your robot must also perform the BumpBot behavior). Specifically, when either of the whiskers is triggered, the robot should reverse for 1 second, and then turn away from the point of contact for 1 second, all while **ignoring any commands from the remote control**. After the robot has finished reversing and turning, it should go back to whatever action it was doing before the impact, and then resume listening for packets sent from its remote.

Once you've completed all of Part 1, you may demonstrate it to your TA now, or you can wait and demonstrate it together with Part 2 (described next).

PROCEDURE PART 2 – FREEZE TAG

Problem Statement

Soon, you are going to be showing your proof-of-concept to upper management. Although the basic remote control operation is working correctly, you feel that your demo should include something that can generate some "real excitement". Having two robots play freeze tag with one another sounds like a good idea!

Specifications

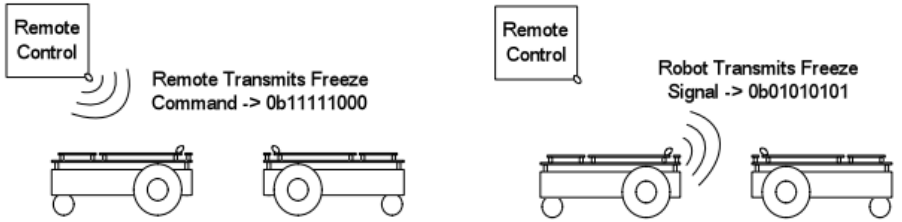
1. Each remote must be able to send a sixth action, "freeze", to its robot. The action code for freeze will be 0b11111000, and will be transmitted using the same packet structure as in Part 1 (i.e., send the robot address first, and then the freeze action code). This transmission, from remote to robot, is depicted in Figure 1a.
2. When a robot receives a packet from its remote containing the freeze action code, the robot must immediately transmit a standalone 8-bit "freeze signal", 0b01010101. This freeze signal, which is sent directly **without any address byte sent first**, is a transmission from one robot to all other nearby robots, as depicted in Figure 1b.
3. Any robot that receives an 8-bit freeze signal should "freeze" for five seconds, **except for** the robot that just sent the freeze signal itself.

   Freezing specifically means: **halting, not responding to whiskers, and not responding to commands or other freeze signals**.
4. When a robot unfreezes, it should immediately resume what it was doing before it was frozen. After being frozen three times, a robot should stop working (i.e., stay frozen) until it is reset.

To fully demonstrate your correct Part 2 (freeze tag) implementation, you will need to work with another team to have two robots compete/freeze each other.

STUDY QUESTIONS / REPORT

A full lab write-up is required for this lab. When writing your report, be sure to include a summary that details **what you did and why, and explains any problems you encountered.** Your write-up and code must be submitted by the end of your final (Week 10) lab session. As usual, NO LATE WORK IS ACCEPTED.



(a) Freeze command (address + action) (b) Freeze signal (from robot)

Figure 1: Freeze command and freeze signal transmission

Study Questions

No additional questions for this lab assignment.

CHALLENGE

Implement the following improvements to your Lab 8 code:

- (3 pts) Use Timer/Counter1 **with interrupts** to implement the 1-second Wait function used within HitRight and HitLeft on the robot. Your completed program must correctly perform a full HitRight/HitLeft routine (move backward for 1 second, then left/right for 1 second) without being interrupted by additional bumper hits or received USART data.
- (7 pts) Combine your Lab 8 functionality with your Lab 6 functionality. Specifically, replace the Halt and Freeze actions that the remote sends with SpeedUp and SpeedDown, and have the robot adjust its speed with Fast PWM based on the received SpeedUp and SpeedDown commands. 16 speed levels and a 4-bit speed indication are required, just as in Lab 6.

Demonstrate your challenge improvements to your TA, and make sure to submit your challenge .asm files separately from the regular Lab 8 files.