Lab2: Performing Concurrent Tasks using Arduino Clocks and Registers

Paria Naghavi (1441396)
Anna Mai (215101)
Assignment: Lab 2 July 14, 2023

Table of Contents

Introduction

This report explores various methods for controlling external LEDs, generating sound using Timer/Counter, and effectively managing concurrent tasks involving LED displays, speaker tones, and joystick input. By adding concurrent task management, the report shows the capability of the Arduino platform to handle interactive operations.

Furthermore, the report highlights the integration of different types of I/O devices. It examines techniques for controlling external LEDs, allowing for precise manipulation of their flashing patterns and behaviors. Additionally, it explores the generation of sound using Timer/Counter, enabling the Arduino to produce tones of different frequencies. The lab also demonstrates the utilization of a joystick as an input device, showing how its movements can be displayed concurrent with tasks, such as controlling LED displays or generating sound.

Methods and Techniques

In Part 1, the report covers different methods of controlling external LEDs. It starts with using the pinMode() and digitalWrite() functions for basic LED flashing patterns. Then, it progresses to direct access of I/O pins using the DDR and PORT registers, providing more control over pin behavior. The lab shows how the DDR register configures pins as inputs or outputs, while the PORT register controls output voltage levels. This technique is demonstrated by manipulating the DDR and PORT registers for LEDs using predefined macros.

Part 2 focuses on generating tones using Timer/Counter. It explains the setup process, including configuring the waveform generation mode (WGM), output compare (OC) behavior, and prescaler value for Timer/Counter4. The report details how the TCCR4A and TCCR4B registers are utilized to control the generation of tones, and the significance of the OCR4A register in determining the frequency and behavior of the speaker output. The Timer/Counter4 prescaler is also discussed, highlighting its role in defining the time period and frequency range of the timer.

In Part 3, the concept of concurrent tasks is introduced. The report describes the sequential execution of tasks A and B, the simultaneous execution of tasks A and B, and the addition of playing "Mary Has a Little Lamb" simultaneously with Task A. It explains the utilization of flags to control task activation and the use of conditional statements to manage task concurrency. The lab exercises emphasize the importance of avoiding the use of delay() function for concurrent tasks and demonstrates how LED cycling and speaker tone generation are achieved based on task states and timing.

Part 4 incorporates input from a joystick and output to an 8x8 LED matrix and a speaker. The report explains the usage of a shift register for controlling individual LEDs and introduces the spiTransfer function for sending SPI commands and data to the LED matrix. It also describes the process of reading joystick coordinates using analog inputs and scaling them to match the LED matrix dimensions. Additionally, it covers the integration of speaker control, including the generation of melodies and beats using arrays, and the utilization of the speakerCycle function.

Experimental Results

Part 1- Hardware Bit Manipulation Introduction

This part provides hands-on demonstrations of hardware manipulation techniques using the Arduino platform. First, we explore different methods to control external LEDs using the Arduino board. We start by using the pinMode() and digitalWrite() functions to achieve a specific flashing pattern. Then, we moved on to direct access to I/O pins using the DDR and PORT registers, which gives us more control over the pins' behavior. Next, we generated sound using Timer/Counter and waveform generation features.

1.2- Flashing External LEDs Using pinMode() and digitalWrite() Functions

In this section we examined accessing I/O pins of the Arduino using pinMode() to achieve the desired LED flashing pattern. Each LED is on for 333ms and off for 666ms. The lab instructions allowed using digitalWrite() and delay() to create the flashing sequence required. The video for this section can be found here.

1.3 and 1.4- Direct Access to I/O Pins Using DDR and Ports

In this section of the code, direct pin access through predefined macros is used to control the I/O pins of the Arduino. This approach achieves the desired LED flashing pattern, which is equivalent to using the pinMode() and digitalWrite() functions provided by Arduino.

The Data Direction Register (DDR) register plays a crucial role in configuring pins as inputs or outputs. Each bit in the DDR register corresponds to a specific pin on the microcontroller. By setting a bit to 1, we configure the corresponding pin as an output. Conversely, setting the bit to 0 configures the pin as an input. For example, DDRD is the DDR register for Port D, setting the 4th bit (DDRD4) to 1 makes Pin 4 of Port D an output, and setting it to 0 makes it an input.

The PORT register controls the output voltage of the pins configured as outputs. Like the DDR register, each bit in the PORT register corresponds to a specific pin. When a pin is configured as an output (according to the DDR register), setting the corresponding bit in the PORT register to 1 will output a high voltage on the pin, while setting it to 0 will output a low voltage. For the Arduino MEGA board, pins 47, 48, and 49 are mapped to port L. Each port has 8 bits (0-7) corresponding to individual pins. In the case of port L, the pins are labeled as L0, L1, L2, L3, L4, L5, L6, and L7. Therefore, pin 47 corresponds to port L, bit 0 (L0), pin 48 corresponds to port L, bit 1 (L1), and pin 49 corresponds to port L, bit 2 (L2). By manipulating the DDR and PORT registers, you can control the direction and behavior of the individual pins on the microcontroller. This allowed to set pins as inputs or outputs and control their voltage levels and consequence flashing rate. The demo for 1.4 is the same as 1.2.

Part 2- Generating a Tone using 16 Bit Timer/Counter

The purpose of this part of the lab is to generate tones of different frequencies using the Timer/Counter4 of the Arduino. It sets up the necessary configurations and functions to control

the generation of tones. First, we define the pin and port macros for the speaker pin. `SPEAKER_PIN` is set to `PH3`, which represents pin 3 on Port H or Pin 6 on the hardware. Next, `SPEAKER_PORT` represents the PORTH register, and `SPEAKER_DDR` represents the DDRH register. From the table 13-24 of the documentation, we can see the PH3 pin's alternate function is being an output compare and PWM output A for the timer 4.

Table 13-24. Port H Pins Alternate Functions

Port Pin	Alternate Function
PH7	T4 (Timer/Counter4 Clock Input)
PH6	OC2B (Output Compare and PWM Output B for Timer/Counter2)
PH5	OC4C (Output Compare and PWM Output C for Timer/Counter4)
PH4	OC4B (Output Compare and PWM Output B for Timer/Counter4)
PH3	OC4A (Output Compare and PWM Output A for Timer/Counter4)
PH2	XCK2 (USART2 External Clock)
PH1	TXD2 (USART2 Transmit Pin)
PH0	RXD2 (USART2 Receive Pin)

In the setup() function, the waveform generation mode (WGM) of Timer/Counter4 is set to Fast PWM mode (Mode 14). Using the table 17-2 of the documentation, the WGM bits (WGM41, WGM40, WGM43, WGM42) in the TCCR4A and TCCR4B registers are set for PWM. TCCR4A primarily deals with waveform generation modes and output compare options, while TCCR4B is focused on setting the prescaler value for Timer/Counter4 using CS40 and CS41 bits.

The Output Compare A (OC4A) is configured to toggle on compare match by setting the COM4A0 bit in the TCCR4A register. The TCCR4A register is an 8-bit register that contains various control bits for Timer/Counter4. Among these bits, there are two bits relevant to OC4A: COM4A1 and COM4A0. These bits determine the behavior of the OC4A pin, which is an output pin associated with Timer/Counter4. The COM4A0 bit, specifically, controls the toggle behavior of the OC4A pin on compare match. When COM4A0 is set to 1, the OC4A pin will toggle its state (high to low or low to high) whenever a compare match occurs. By manipulating the COM4A0 bit in the TCCR4A register, we can control the output behavior of the OC4A pin. This allows you to generate waveforms, such as PWM signals or trigger events based on compare match, using the OC4A pin. Please see below for tables 17.11.3 and 17.11.7 showing both timer 4 control registers.

17.11.3 TCCR4A – Timer/Counter 4 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0xA0)	COM4A1	COM4A0	COM4B1	COM4B0	COM4C1	COM4C0	WGM41	WGM40	TCCR4A
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

17.11.7 TCCR4B – Timer/Counter 4 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0xA1)	ICNC4	ICES4	–	WGM43	WGM42	CS42	CS41	CS40	TCCR4B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Timer/Counter4 prescaler is set to 64 by setting the CS41 and CS40 bits in the TCCR4B register. This determines the frequency range of the timer. In the TC4 module, CS stands for Clock Select. The CS bits in the TCCR4B (Timer/Counter4 Control Register B) register control the prescaler of Timer/Counter4, which determines the clock source and division factor for the timer. By setting the CS41 and CS40 bits in the TCCR4B register, the code is selecting a prescaler value of 64 for Timer/Counter4. This means the timer will count with a frequency that is divided by 64 compared to the main clock frequency of the microcontroller. The prescaler setting determines the time period and frequency range of the timer.

This section we demonstrated how to control the tone generation and play different frequencies with specified durations. In the void loop() function, three tones with different frequencies are played. The playTone() function is called with the frequency and duration as arguments. Each tone plays for 1 second with a delay of 1000 milliseconds between each tone. By dividing the system clock frequency by 64 times the desired frequency, the code calculates the duration of each cycle of the waveform. This duration is stored in the period variable and determines the timing for generating the tone using Timer/Counter4. Please see the demo here.

Part 3- Concurrent Tasks

In this part we explore concurrent tasks, the speaker tone and LED flashing. During the first section of part III, task A is defined to turn each of the 3 LEDs for 333 milliseconds and off the rest of the time. Task B plays tones at 400 Hz, 250 Hz and 800 Hz frequencies for 1 second each. Task C manages the timing of Task A and B.
Part 3.1) Flashing Then Tone

3.1- Task C: Tasks A and B Sequentially

In this section of part 3, we considered the first task for 2s and B for 4s and one second delay between A and B repeating. Task C has to manage the execution of task A and B sequentially. Task A is the same as 1.4 and task B is the same as 2.4. They both accept a parameter indicating their time durations for each task. It is important to note that in this section of Part III, we could easily use delay() in Task A because the tasks were executed sequentially and concurrency would not be interrupted. This will change in the next two section of Part III. Please see the demo for 3.1 here.

3.2- Task C: Tasks A and B Simultaneously

The second section of Part III has a similar start as 3.1; however, task C requires task A and B to happen simultaneously for 10 seconds. In order to achieve this objective, we modify the code from the previous section by adding taskAActive and taskBActive flags. Since there is concurrency between task A and B, we cannot use delay() function, because it will halt the

execution of the program for parallel tasks. Instead we used conditional statement to investigate what state each task is in any given time.

In 3.1, the code organizes the tasks as separate functions (taskA(), taskB(), and taskC()) and executes them within the loop() function based on predefined durations. In 3.2 implementation, we integrated the task logic directly into the loop() function. Task A handles LED cycling, and Task B manages the speaker cycle. The execution of tasks is controlled by toggling the state of corresponding flags (taskAActive and taskBActive). Instead of separate functions, the task code is directly implemented within the loop() function. The demo can be found here the start of the cycle is at 10s into the video.

3.3- Task C: Tasks A and Play "Mary Has a Little Lamb" Simultaneously

The third section of Part III has a similar start as 3.2; however, task B is required to play "Mary Has a Little Lamb" simultaneously with task A. The logical conditionals used in 3.2 remains the same for this section, however the speaker cycle changes to play the melody and beats.
The speakerCycle function is responsible for generating the frequency to be played by the speaker based on the current time and the state of Task B. It uses an array, melody, and an array of beats to define a musical note. It calculates the duration of each note in the melody by multiplying the beat value with a fixed duration. Then, it retrieves the frequency value freq corresponding to the current note in the melody. The function checks the state of Task B (taskBActive flag) to determine whether to play the tone or not. If Task B is active, it proceeds with generating the frequency based on the melody and beats. If Task B is not active, it returns a frequency of 0, indicating no sound should be played. For timing and indexing, it uses a static variable currentTime and an index variable noteIndex to keep track of the progress in the melody. These two values are updated when the note duration exceeds a certain threshold, allowing for sequential playback of the melody. The function returns the calculated frequency freq to be used in the `loop` function for setting the tone of the speaker. Please see the demo video for this section here.

Part 4- Concurrent Tasks using a Joystick, an LED Display, and a Speaker

4.1 - Input: Joystick, Output: 8x8 LED register and Arduino's Shift Register

In this section we provide input using a joystick and output using 8x8 LED matrix display. The movement of the joystick should be shown on the display with stable lit dots.
To control each LED individually, it required a significant number of microcontroller pins, especially if each LED is to be controlled directly. The shift register acts as a parallel-to-serial converter, taking in the digital data representing the LED states and converting it into a serial stream. The shift register is represented by the spidata array, which is used to transfer SPI commands and data to the LED matrix.
The spiTransfer function is responsible for transferring SPI commands and associated data to control an LED matrix. It initializes a shift register array, sets the opcode and data values, clears the chip select pin, and sequentially shifts out the data to the LED matrix using the shiftOut function. Finally, it sets the chip select pin to indicate the completion of data transmission.
The findCoordinates function has two analog readings performed using the analogRead function to read the voltage levels from the x and y pins of the joystick. These readings provide

values ranging from 0 to 1023, representing the position of the joystick along the respective axes. The x and y coordinates are calculated by dividing the analog readings by 128, which scales the values to a range of 0 to 7 (since 1023 divided by 128 is approximately 7). These scaled values represent the position of the joystick within an 8x8 LED matrix. The coordinates array is returned from the findCoordinates function and used within the loop function for further processing or controlling the LED matrix based on the joystick's position. The loop function continues to execute in a loop, repeatedly calling findCoordinates and performing subsequent actions based on the returned coordinates. Please see demo here.

4.2- Input: Joystick, Output: 8x8 LED register and Speaker

This section is very similar to 4.1 with addition of concurrent task of playing "Mary Has a Little Lamb" on the speaker. There are few additions: First, new constants are defined for additional control operations, pins, and note frequencies. The speaker-related functionality is also introduced, including the speakerCycle function, which plays a predefined melody
In the setup function, the configuration of Timer/Counter4 and the speaker pin are added to enable sound generation. The spiTransfer function remains largely the same, transferring SPI commands to the LED matrix. However, the findCoordinates function has been modified to adjust the x-axis reading and utilize the full range of the joystick.

In the loop function, before updating the LED matrix, a for loop is introduced to clear the board by setting all rows to zero. The OCR4A register is updated to control the frequency of the speaker's sound output based on the freq value returned by speakerCycle. Overall, the updated code adds functionality for sound generation using a piezo speaker and clears the LED matrix before updating its state. Please see the video here.

Code Documentation

1.2. Flashing External LEDs using pinMode() and digitalWrite() functions

This section was very similar to Lab 1. More details on how the frequency of flashing can be manipulated using delay() and digitalWrite() functions can be found in Lab 1 report.

1.3 and 1.4. Direct Access to I/O Pins Using DDR and Ports

This section was very similar to Lab1. More details on how the frequency of flashing can be manipulated using delay() and digitalWrite() functions can be found in Lab 1 report. The code for this section includes a setup() function using `DDR_L |= (1 << PIN_47) | (1 << PIN_48) | (1 << PIN_49);` , which sets the DDR of Port L bits to 1, indicating that pins 47, 48, and 49 are set as outputs.
In the loop() function. The `|=` operator is used in the code to perform a bitwise OR operation and assign the result back to a variable. It is used to set specific bits in the DDR_L and PORT_L registers, which control the direction and state of the pins, respectively. For example, `PORT_L |= (1 << PIN_47);` sets the bit for PIN_47 to 1, turning on LED 47 by setting the corresponding pin in Port L to high. On the other hand `&=` operator was used to turn the LED. By the following command, `PORT_L &= ~(1 << PIN_47);`, PIN_47 was set to 0. Similar operations are performed for LED 48 and LED 49, sequentially turning them on and off. The delay() function was used to introduce a delay of 333 milliseconds between each LED state change, creating the desired

flashing pattern.

Part 2

This section demonstrates how to control the tone generation and play different frequencies with specified durations. In the void loop() function, three tones with different frequencies are played. The playTone() function is called with the frequency and duration as arguments. Each tone plays for 1 second with a delay of 1000 milliseconds between each tone. It calculates the period of the tone by dividing the system clock frequency (`F_CPU`) by 64 times the desired frequency.

The bit_set() function is called to set the speaker pin as an output by setting the corresponding bit in the SPEAKER_DDR register. It performs a bitwise OR operation between the register value (`reg`) and a bitmask created by shifting `1` to the left by the `bit` position. This effectively sets the corresponding bit to 1 in the register. The function bit_clear() does the opposite using bitwise AND operation. Inside the while loop, the compare match value (`OCR4A`) is set to half the period. This determines the point at which the timer will generate a match interrupt. By setting it to half the period, we achieve a 50% duty cycle for the generated waveform. The delayMicroseconds() function is used to introduce a delay based on the period of the tone, allowing the tone to play for the desired duration. Finally, after the loop completes, the speaker is turned off by setting the compare match value to 0.

the OC4A (PH3) pin on the microcontroller can serve as an output for the Output Compare A function of Timer/Counter4. By configuring it as an output and utilizing PWM mode, you can generate PWM signals or trigger events based on comparison matches. The bits on timer registers control how the timer counts, as well as the actions taken when certain conditions are met.

3.1- Task C: Tasks A and B Sequentially

The code for this section is the reuse of the same functions as 1.4 and 2.4 with the addition of taskC() function which performs task A, B and 1 second of delay. Since taskC() is a void while loop, it will iterates indefinitely.

3.2- Task C: Tasks A and B Simultaneously

The loop() function orchestrates the execution of different tasks and controls the behavior of the LEDs and speaker based on the active task flags and current time. Task A is started by setting taskAActive flag to true. The current time is recorded using millis() function. A while loop runs until the specified taskATime duration is reached. Within the loop, the ledCycle() function is called to control the LED cycle for Task A. After the specified duration of Task A, the taskAActive flag is set to false to stop Task A. Task B is started by setting taskBActive flag to true. A while loop runs until the specified taskBTime duration is reached. Within the loop, the speakerCycle() function is called to determine the frequency for the speaker based on the current time. The OCR4A register is updated accordingly to play the tone. After the specified duration of Task B, the taskBActive flag is set to false and the OCR4A register is set to 0 to stop the speaker.

For the simultaneous execution of task A and task B, we set both taskAActive and taskBActive flags to true. The current time is recorded using millis() function. A while loop runs until the specified taskABTime duration is reached. Within the loop, both the LED cycle (ledCycle()) and speaker cycle (speakerCycle()) functions are executed. The OCR4A register is updated based on the current time and speaker frequency. After the specified duration of simultaneous execution, both taskAActive and taskBActive flags are set to false, and the OCR4A register is set to 0 to stop the speaker. A delay() function is used to introduce a downtime interval of downtime duration.

The speakerCycle() function determines the frequency for the speaker based on the current time and the state of taskBActive flag. It returns the frequency value to be used in the main loop. The ledCycle() function controls the LED cycle based on the state of taskAActive flag. It uses the bit_set() and bit_clear() functions to set or clear specific bits in the LED_PORT register to control the LEDs.

3.3- Task C: Tasks A and Play "Mary Has a Little Lamb" Simultaneously

This section aims to create a system that combines LED flashing and speaker tone generation using an Arduino board. After the pin and register definitions and global flags for task activation process, the setup function initializes the pin configurations and sets the necessary registers for LED control, PWM generation, and speaker control. Similar to previous section, the speaker pin is configured as an output using the bit_set function. The loop function defines the timings and sequences of Task A and Task B, and their combination. It begins by starting Task A and executing the LED cycle for the specified taskA time duration. Then, Task B is started, and the speakerCycle function is used to generate tones based on the current time. Next, task A and Task B are then run simultaneously for the specified taskAB time duration, with LED cycling and tone generation. Finally, there is a delay for the downtime duration before repeating the loop.

4.1- Input: Joystick, Output: 8x8 LED register and using Arduino's Shift Register

In the code, the spiTransfer function is responsible for loading the data into the shift register. It takes two parameters: opcode and data. The opcode represents the control operation or command for the LED matrix, and the data represents the data associated with that command. Within the spiTransfer function, the spidata array is initialized to zeros using a for loop. Then, the opcode and data values are assigned to specific elements of the spidata array. After loading the data into the shift register, the code uses the shiftOut function to shift out the bytes of data to the LED matrix, starting with the leftmost bit. The shiftOut function uses the DIN and CLK pins to perform the shifting operation.

After the constant for the LED monitor is set up, void spiTransfer(volatile byte row, volatile byte data); is defined. The volatile keyword is a type qualifier that informs the compiler that the value of a variable may change at any time, even if it is not modified within the current scope. The pins, port and DDR of the 8x8 LED matrix are defined with pin 47,48 and 49 pins on port L connected to DIN, CS and CLK on the LED device. Similarly, the joystick pins, port and DDR are defined. The VRx and VRy are connected to A0 (PF0) and A1 (PF1) which are on port F managed

by DDRF. The spidata is an array of bytes used as a shift register for SPI communication. It has a length of 2, where the first byte represents the control byte and the second byte represents the data byte. The code bit_set(BOARD_PORT, CS_PIN); sets a specific bit of the BOARD_PORT register to 1, effectively setting the corresponding pin to a high logic level. The Chip Select (CS) pin of the LED matrix to a high logic level. The CS pin is typically used in SPI communication to select the device with which the microcontroller wants to communicate. By setting the CS pin to a high logic level, the LED matrix is selected, indicating that it is ready to receive commands or data from the microcontroller.
The findCoordinates function reads the analog data from the joystick's X and Y axes using analogRead. It then scales the readings by dividing them by 128 to obtain coordinates ranging from 0 to 7, which corresponds to an 8x8 LED matrix. The x and y coordinates are stored in the coordinates array, which is dynamically allocated, and then returned. This code enables the control of an LED matrix based on the joystick input, allowing for interactive visual output.

4.2- Input: Joystick, Output: 8x8 LED register and Speaker
This section's code extends the functionality of 4.1 by incorporating speaker control and playing tones in addition to controlling the LED matrix and reading joystick coordinates. Control for the speaker was setup, using PWM and generate different frequencies. The TCCR4A and TCCR4B registers are configured to set the waveform generation mode, output compare, and prescaler for the speaker. Next, the OCR4A register is adjusted to control the frequency of the speaker output and the SPEAKER_DDR is set to configure the speaker pin as an output.
For playing "Mary Has a Little Lamb" simultaneous, melody and beats arrays were defined to store the note frequencies and durations for playing a melody. The speakerCycle() function is added to iterate through the arrays and play the melody.

The setup() function now includes the configuration for the speaker and adjusts the pins and registers for the LED matrix and joystick, similar to part 3.3. The loop() function incorporates the new functionalities by calling the findCoordinates() function, setting the row control for the LED matrix, and adjusting the speaker frequency based on the melody.
This code adds speaker control for playing "Mary Has a Little Lamb", while controlling the LED matrix and reading joystick coordinates.

Overall Performance Summary
Our demo videos hyperlinked throughout the report.

Teamwork Breakdown
This lab was completed in collaboration of Paria Naghavi and Anna Mai.

Discussion and Conclusions
The most challenging part of this lab was getting the tasks to run concurrently in part 3.2. It took a different mindset that focused on the states of each task in relation to the current time rather than hard programming the state change after a given amount of time. We are proud of the solution we came up with, but after progressing further into the lab there could've been a cleaner way to program the scheduler. The scheduler could've been implemented in a similar

method to speakerCycle() for part 3.3, where multiple arrays are incremented through. So the scheduler would have arrays that control the global flag for tasks as well as one for the duration for the desired run times. Another thing we could do for future labs is to set up a .h file so that the overhead to maintain our code between each lab part is reduced. This will allow for more organization and usability of code.

Reference

1. ChatGPT
2. C programming language documentations: https://en.cppreference.com/w/c/preprocessor/conditional
3. Lab provided resources