Reporte de la Práctica 2: Ensamblador, Interrupciones & Temporizadores

Andrea Marín Alarcón
Luis Felipe Landa Lizarralde
Miguel González Borja

Resumen—Los lenguajes de alto nivel, al ser compilados, aunque sean eficaces, tienden a perder eficiencia. Esta pérdida tiene un impacto al momento de ejecución de un programa, pues se utiliza una mayor cantidad de recursos a los estrictamente necesarios y el tiempo de ejecución se prolonga. A su vez, se busca que código específico se ejecute en eventos predeterminados que tomen precedencia sobre el comportamiento normal del programa. Por último, la ejecución de código de forma periódica es una característica deseable cuando se busca que un microcontrolador controle un elemento externo. El código en ensamblador, las interrupciones y los temporizadores implementados en el ATmega320 presentan una alternativa de bajo nivel para paliar las deficiencias anteriores. Se desarrollaron tres programas básicos en ensamblador: un cálculo de discriminante, un cálculo de un promedio y uno que hace parpadear a un LED. Dos programas implementaron interrupciones: un LED parpadeante a 2 *Hz* y un contador activado por un fotodiodo. Los temporizadores fueron utilizados para nuevamente hacer parpadear un LED y para implementar un semáforo. Los programas son los mismos resultados de la práctica.

1. INTRODUCCIÓN

En el diseño e implementación de un programa, no solo se busca que este sea capaz de llevar a cabo su función, sino que también lo logre de la manera más eficiente posible. Con esto nos referimos a que el programa se ejecute en el menor tiempo posible y utilizando la menor cantidad de recursos. En lenguajes de alto nivel como Java y Python, esta eficiencia se puede estimar analizando la cantidad de evaluaciones y ciclos implementa el programa. Estas estimaciones permiten comparar la eficiencia de dos programas escritos en el mismo lenguaje, pero difícilmente resulta en una evaluación cuantitativa de los recursos del microprocesador que utiliza cada programa.

El lenguaje ensamblador permite un análisis tremendamente más preciso de costos y eficiencia en un programa. Como el lenguaje de ensamblador es específico a cada microprocesador, hay una mayor cantidad de información de los recursos que ocupa cada instrucción. Además, como a cada instrucción corresponde a un registro de memoria, un programador es capaz de saber de manera exacta la cantidad de memoria que ocupa su programa. De igual manera, en las especificaciones de cada microprocesador se detalla cuántos ciclos de reloj ocupa cada instrucción, con lo que se puede calcular el número total de ciclos que toma ejecutar un programa. Una vez calculado el número total de ciclos, conociendo la frecuencia a la que opera el microprocesador, es un procedimiento trivial calcular el tiempo de ejecución del programa. Con ensamblador, el programador tiene una cantidad mínima de abstracción del código máquina y por ende es capaz de ejercer un mayor control sobre las operaciones realizadas. Es decir, de esta manera, el programador puede asegurarse de ejecutar únicamente las instrucciones fundamentales para la ejecución de su código en vez de depender de un compilador que al carecer conocimiento específico del programa puede autogenerar código que solamente entorpece al programa.

De la misma manera que el lenguaje ensamblador es específico para cada familia de microprocesadores, también lo son el sistema con el que se implementan las interrupciones y los temporizadores. La variedad de interrupciones y temporizadores disponibles en un microcontrolador otorgan una gran gama de posibles combinaciones y utilidades. Se pueden utilizar como contadores, generadores de ondas o disparadores de rutinas específicas a situaciones particulares. Esta misma diversidad de modalidades que presentan las interrupciones junto con los temporizadores permiten a un microcontrolador ejecutar código en intervalos de tiempo determinados por el programador.

La práctica tiene como objetivo ahondar en los temas expuestos anteriormente: comparar un programa diseñado en ensamblador con código auto-generado por un compilador de un lenguaje de alto nivel, explorar usos de interrupciones en varios escenarios y por último combinar los dos temas anteriores con el uso de temporizadores. Los pequeños programas desarrollados fungen como ejemplos didácticos que indican usos futuros de mayor complejidad.

En el Marco Teórico se profundizan los temas del lenguaje ensamblador, las interrupciones y los temporizadores junto con sus configuraciones posibles. La sección de Desarrollo indica a mayor detalle los programas implementados en C y ensamblador. El apartado de Resultados analiza los efectos obtenidos de los programas y resalta sus elementos fundamentales. La sección de Conclusiones detalla los aspectos de la práctica que cada miembro del equipo consideró más relevantes.

2. MARCO TEÓRICO

Todas las computadoras cuentan con un microprocesador que administra las operaciones aritméticas, lógicas y de control del microcontrolador. Cada microprocesador posee su propio conjunto de instrucciones con las que puede realizar diversas operaciones, como: mover datos de un registro a otro, sumar los valores de dos registros, obtener el valor de una dirección de memoria dada, entro otras. Estas instrucciones se encuentran en código máquina, es decir, son cadenas de 0's y 1's por lo que intentar programar directamente en código máquina es tedioso y sería muy fácil cometer errores.

El lenguaje ensamblador está formado por mnemónicos que corresponden a las diferentes instrucciones del microprocesador. Por ejemplo, si `10110` significa 'mover el valor", entonces en lenguaje ensamblador tenemos el comando `MOV` que hace lo mismo [1]. Se dice que es de bajo nivel porque existe una correspondencia (casi) 1-1 entre el lenguaje ensamblador y el código máquina.

Tanto el lenguaje ensamblador, como lenguajes de alto nivel (C++, Java, Python, etc.), necesitan de un traductor para traducir el código fuente a código máquina. Sin embargo, los lenguajes de alto nivel están formados por abstracciones de operaciones de bajo nivel, lo cual permite que el programador se enfoque más en lo que quieren hacer y no en cómo debe de hacerlo la máquina [2]. Es por esto que la mayor ventaja de trabajar con lenguaje ensamblador es que se tiene un control preciso sobre las tareas que realiza el microprocesador. Además, los programas creados en ensamblador se ejecutan más rápido y ocupan menos espacio que aquellos creados en lenguajes de alto nivel [3]. No obstante, el lenguaje ensamblador no es portátil ya que depende de la arquitectura y el conjunto de instrucciones del microprocesador que se utiliza. En esta práctica se realizaron varios sistemas en el lenguaje ensamblador del ATMega256.

También se utilizaron temporizadores e interrupciones en la implementación de algunos sistemas. Un temporizador, o *timer/counter*, es una pieza de hardware del Arduino la cual funciona como un reloj y se puede usar para medir eventos. Todos los microcontroladores cuentan con un reloj interno, así el temporizador es básicamente un contador de los pulsos del reloj interno [4]. Entonces, por ejemplo, si el reloj interno del Arduino tiene una frecuencia de 8MHz, el temporizador cuenta cada 0.125 microsegundos.

Si no queremos que cuente tan rápido, existe una opción de pre-escalamiento, donde se divide la frecuencia del reloj interno entre 8, 64, 256 o 1024 [5]. Regresando al ejemplo anterior, si ahora hacemos un preescalamiento de 256, el temporizador cuenta cada 0.032 milisegundos, pues el temporizador se incrementa cada 256 ciclos de reloj y tenemos que $\frac{1}{8M} \times 256 = 0{,}032m$.

También se puede configurar el temporizador para que cuente señales externas. Esta señal se capta mediante un pin del Arduino y el temporizador incrementa ya sea en el flanco descendente o en el flanco ascendente. El límitedel temporizador está determinado por el número de bits del registro del contador asociado, así, un temporizador de 8 bits cuenta hasta 256 y uno de 16 bits hasta 65,535. El ATMega256 cuenta con 4 temporizadores de 16 bits y 1 de 8 bits [6].

Los temporizadores tienen diferentes registros asociados, los que se utilizaron fueron:

- *TCNTn (Timer/Counter register)*: Donde se guarda el valor actual del contador.
- *TCCRn (Timer/Counter Control Register)*: Se configura el modo del temporizador, el preescalamiento y otras opciones para el PWM.
- *OCRn (Output Compare Register)*: Se utiliza para generar interrupciones o para compararlo con el contador en el modo CTC.
- *TIMSK (Timer Interrupt Mask)*: Se utiliza para controlar el tipo de interrupción a realizar.
- *TIFRn (Timer Interrupt Flag)*: Es el registro que contiene las diferentes banderas que se activan dependiendo del modo del contador. Las que se utilizaron fueron: *TOVn*, que indica cuando ocurre un *overflow* y *OCFn*, que indica cuando el valor del contador (*TCNTn*) es igual al del registro *OCRn*

Es importante mencionar que todos los registros son de 8 bits, así que cuando se tiene un temporizador de 16 bits, tanto el *TCNTn*, como el *OCRn*, se descomponen en dos registros, *high* y *low* [6], los cuales en conjunto forman un número de 16 bits.

Como ya se mencionó, los temporizadores tienen diferentes modos: modo normal, modo CTC (*Clear Timer on Compare*), modo PWM (*Pulse Width Modulation*) y modo PWM rápido. En esta práctica sólo se utilizaron los primeros dos.

Para explicar mejor los diferentes modos, supondremos que estamos trabajando con un temporizador de 8 bits. En el modo normal, el temporizador cuenta hasta su valor máximo, 0xFF, y se activa la bandera *TOVn* cuando pasa de 0xFF a 0x00 [5]. Para modificar la frecuencia con la que se prende *TOVn*, se puede cargar un valor inicial al registro *TCNTn* y así no tienen que pasar 256 ciclos de reloj para que se active *TOVn*. Por ejemplo, si se inicializa *TCNTn* en 206, al temporizador sólo le tomará 50 ciclos para que se active *TOVn*, pues en 49 ciclos *TCNTn* llega a su valor máximo y en el siguiente ciclo, (*TCNTn* = 256) se activa *TOVn*.

El otro modo del temporizador se llama *Clear Timer on Compare* o CTC. En este modo, el temporizador cuenta hasta que el valor de *TCNTn* es igual al del registro *OCRn*, y en ese momento se activa la bandera *OCFn* y se borra el contador, es decir $TCNTn = 0$. Si quisiéramos hacer lo mismo que en el ejemplo anterior, pero en el modo CTC, entonces se inicializa *OCRn* en 50. Así, el temporizador cuenta 50 ciclos y luego se reinicia.

Por último, los temporizadores también pueden ser usados para crear interrupciones. Las interrupciones se activan cuando se cumplen ciertas condiciones. Cuando se activa la interrupción el AVR se para, guarda la ejecución de la rutina principal, ejecuta el código de la interrucupción (el cual se encuentra en un método llamado *Interrupt Service Routine*, ISR) y, posteriormente regresa a la rutina principal [7]. La ventaja de las interrupciones es que no se necesita checar continuamente si cierta condición se cumple para poder ejecutar un pedazo de código. Por ejemplo, si la rutina principal tiene un *delay*, pero al mismo tiempo hay una señal de entrada que se debe registrar cada que se reciba, sin importar si se está en *delay* o no, entonces se activa

una interrupción para esta señal. A esto se le llama una interrupción externa.

Existen otros tipos de interrupciones. La interrupción de *overflow* se activa cada que el temporizador llega a su valor máximo y se prende la bandera *TOVn*. La interrupción de comparación es aquella que se activa cada que el valor de *TCNTn* es igual al de *OCRn*. Estos dos tipos de interrupción se configuran en el registro *TIMSK*.

Por último, al implementar las interrupciones en la práctica se utilizó la señal de un sensor infrarrojo, el cual está compuesto por: un LED IR, el emisor, y un *fotodiodo*, que funciona como receptor. El LED IR es un LED que emite rayos infrarrojos con una amplitud de onda desde 700nm hasta 1 mm. Los LEDs IR generalmente están hechos de arseniuro de galio o arseniuro de galio de aluminio [8]. La luz infrarroja se encuentra debajo de la parte del espectro de luz que es visible para el ojo humano. El *fotodiodo* es un tipo de detector de luz, está basado en un diodo de unión PN y se utiliza para convertir luz en corriente o voltaje [9]. El voltaje de salida del *fotodiodo* es directamente proporcional a la cantidad de luz infrarroja que recibe. Esto se debe a que la corriente se genera por los fotones que absorbe el *fotodiodo*.

3. DESARROLLO

3.1. Ensamblador

Para los siguientes programas se escribió un código en C y un código en lenguaje ensamblador dentro del entorno de desarrollo de Arduino.

Discriminante

El código implementado calculó el valor del discriminante para una ecuación cuadrática de la forma

$$ax^2 + bx + c = 0$$

La fórmula utilizada fue:

$$d = b^2 - 4ac$$

Además, se hizo un circuito simple con un LED, el cual se prende si el resultado anterior es positivo.

Promedio

En ambas implementaciones se sumó el valor de n números y posteriormente se dividió el resultado entre n. El algoritmo escrito en ensamblador realiza una división entera y regresa el residuo. Por otro lado, el algoritmo de C realiza la división con decimales.

Entradas y salidas

LED que parpadea: Para el código en C, se utiliza la función *delay* para hacer el LED alternar de estado cada 250 ms.

En ensamblador, se configuró el *Timer1* en modo *CTC* para contar 250ms (*OCR1A = 0x2000*), generando una interrupción cuando la comparación se cumple, la cual alterna el estado del LED.

LED con botón: Para C, se utiliza una bandera booleana para voltear el estado del LED cada vez que se presiona el botón. Luego se deja un retardo de 250ms para evitar que el LED cambie de estado varias veces cuando se acciona el botón.

En ensamblador, se tomó el bóton como la interrupción externa 1 (*INT1*), cambiando el estado del LED cada vez que se genera la interrupción.

3.2. Interrupciones

Ambos códigos se hicieron en C.

3.2.1. *Contador con botón*

En este sistema se hizo parpadear un LED con un algoritmo similar al de 3.1. Sin embargo, después de apagar el LED y esperar 250ms, se revisa la entrada de un botón, y en caso de que esté presionado, se incrementa el contador.

3.3. Contador con sensor IR

Para este sistema también se implementó un contador junto con un LED que parapadea cada 250 ms, pero, a diferencia del anterior, el contador se incrementa con la entrada de un sensor IR.

El código utilizado para implementar las interrupciones se escribió en C. Para configurar el dispositivo, primero se apagó la bandera de interrupciones globales para no interrumpir el proceso de configuración. Se utilizó un pin análogo para la lectura del valor del *fotodiodo*. Este pin análogo, corresponde al PINCT16 por lo que se activó la interrupción *PCIE2 (Pin Change Interrupt Enable)* en el registro *PCICR (Pin Change Interrupt Control Register)*. Además, se activó el *PCINT16* en la máscara de interrupciones asociadas a los pines, *PCMSK2*. Al finalizar la configuración se vuelve a activar la bandera de interrupciones globales.

En la rutina de servicio asociada a la interrupción, se incrementa el contador global del programa.

3.4. Temporizadores

Para todos los sistemas se utilizó el *Timer1*, que es de 16 bits y la configuración de los modos del temporizador se realizó en el *TCCR1B*.

3.5. Modo normal

Dado que el reloj interno del ATMega2560 es de 16MHz, para poder bajar la frecuencia a 2Hz, se hizo un preescalamiento de 1024 y posteriormente se calculó el valor inicial de TCNT1. Para hacer parpadear el LED se utilizó una interrupción de *overflow*, *TIMER1_OVF*, así, cada que la bandera *TOV* se activa, el LED cambia de estado. Además, el código para realizar esta acción se hizo en lenguaje ensamblador.

3.6. Modo CTC

Dado que el reloj interno del ATMega2560 es de 16MHz, para poder bajar la frecuencia a 4Hz, se hizo un preescalamiento de 1024 y posteriormente se calculó el valor a almacenar en *OCR1A*.

Al igual que en el modo normal, el parpadeo del LED se implementó en lenguaje ensamblador y se controló mediante una interrupción. Sin embargo, al estar en modo CTC, la interrupción utilizada fue de comparación, *TIMER1_COMPA*.

3.7. Semáforo

Dado que el reloj interno del ATmega2560 es de 16MHz y los temporizadores son de máximo 16 bits, solo se puede contar hasta un máximo de 4 segundos. Con esto en consideración, se configuró el temporizador 1 para contar 1 segundo.

En este caso no se utilizaron interrupciones. Toda la lógica del semáforo se implementó con un código en lenguaje ensamblador, el cual enciende el LED rojo y apaga el verde y el amarillo; luego, cuenta 10 segundos con el LED rojo encendido. Después de los 10 segundos, apaga el LED rojo y prende el verde, cuenta 12 segundos y finalmente prende el LED amarillo y cuenta 3 segundos con el LED verde y amarillo encendidos.

4. RESULTADOS

4.1. Código de ensamblador auto-generado

Para cada uno de los siguientes problemas, se comparó el código autogenerado, basado en C, contra código escrito en ensamblador. El código relevante puede ser encontrado en el repositorio.

4.1.1. Cálculo de discriminante

Para el código en C, se declararon 3 variables locales dentro del *loop* principal, *a = 10*, *b = 20* y *c = 30*, y luego se calculó la variable local *det = b*b-4*a*c*. Si el valor de *det* es mayor a 1, se enciende un LED.

En el código de ensamblador se cargaron los valores de *a*, *b* y *c* de manera inmediata en los registros R18, R16 y R19 respectivamente. Luego, se guarda *R17:R16* $\leftarrow$ *b * b* y *R19:R18* $\leftarrow$ *a * c*. Finalmente, se recorre R19:E18 2 veces a la izquierda y se hace la resta a 2 bytes para obtener el discriminante.

A diferencia del código escrito, el código autogenerado realiza el calculo del discriminante una sola vez, dejando constante el estado del LED.

La configuración de los pines se encuentra en el Cuadro 1.

4.1.2. Cálculo de promedio

Para el código de C, se realiza la lectura de los datos a través del puerto serial. Estos se van acumulando en una variable local (*sum*), y luego se divide entre el total de números (*n*) para conseguir el promedio.

Por otro lado, en ensamblador se considera un arreglo de números fijos. Para sumar los números se utiliza un apuntador a los valores del arreglo en memoria y se almacena la suma parcial en R16. Una vez que se tiene la suma total, se resta *n* de la suma hasta que esta sea menor a *n*, y se guarda el total de vueltas (la división entera) en R20.

El código autogenerado utiliza rutinas pre-hechas para la lectura de números y para la división en punto flotante.

No se utilizaron pines para este sistema.

4.1.3. LED parpadeando

El código autogenerado utiliza una rutina para contar 250ms cada vez que se llama la función *delay*, que a su vez cuenta en microsegundos por medio del *Timer0* hasta llegar al valor deseado.

La configuración de los pines se encuentra en el Cuadro 1.

4.1.4. LED con botón

El código autogenerado lee el valor de entrada y escribe el estado del LED correspondiente. También utiliza la función *delay* como se describió en 4.1.3.

La configuración de los pines se encuentra en el Cuadro 1.

4.2. Interrupciones

4.2.1. Contador sin Interrupciones

El programa implementado no incrementa el contador si el Arduino estaba ejecutando la instrucción *delay*, sin importar si se presionaba o no el botón.

La configuración del sistema puede encontrarse en el Cuadro 3.

4.2.2. Contador con Interrupciones

Se escribió un programa en C que hiciera el LED parpadear cada 250ms por medio de la función *delay* dentro de un ciclo infinito. Además, se conecto un fotodiodo como interruptor externo, y cuando su valor cambiaba, ocasionaba la rutina de servicio asociada al *Pin Change Interrupt 2* que incrementaba el contador. En este caso, el contador incrementa sin importar el estado del ciclo infinito que hace al LED parpadear.

La configuración del sistema y su imagen pueden encontrarse en el Cuadro 4 y la Figura 1.

4.3. Temporizadores

4.3.1. Modo Normal

Se calculó que con un preescalamiento de 1024, se necesitaba un valor inicial en TCNT1 de:

$$2Hz = \frac{16MHz}{1024 * (2^{16} - TCNT1)} \Rightarrow TCNT1 = 57,536$$

La configuración del sistema se puede encontrar en los Cuadros 1 y 6.

4.3.2. Modo CTC

Dado que el reloj interno del ATmega2560 es de 16MHz, se calculó que con un preescalamiento de 1024, se necesitaba un valor en OCR1A de:

$$4Hz = \frac{16MHz}{1024 * (OCR1A)} \Rightarrow OCR1A = 4,000$$

La configuración del sistema se puede encontrar en los Cuadros 1 y 6.

4.3.3. Semáforo

Para contar 1 segundo con el temporizador, se calculó que el valor de OCR1A es:

$$1Hz = \frac{16MHz}{1024 * (OCR1A)} \Rightarrow OCR1A = 16,000$$

Después, se cuentan 10 segundos con el LED rojo encendido, 12 con el verde encendido, y 3 con el verde y el amarillo encendidos.

La configuración del sistema se puede encontrar en los Cuadros 5 y 6. Una imágen se puede observar en la Figura 2.

Pin	I/O	Elemento
13	Output	LED

Cuadro 1
Configuración de pines para los sistemas 4.1.1, 4.1.3, 4.3.1 y 4.3.2

Pin	I/O	Elemento
13	Output	LED
31	Input	Botón

Cuadro 2
Configuración de pines para el sistema 4.1.4

Pin	I/O	Elemento
13	Output	LED
20	Input	Botón

Cuadro 3
Configuración de pines para el sistema 4.2.1

Pin	I/O	Elemento
13	Output	LED
A8	Input	Fotodiódo

Cuadro 4
Configuración de pines para el sistema 4.2.2

5. CONCLUSIONES

En cuanto a la diferencia entre el código autogenerado basado en C y el código escrito en ensamblador, en general no se observa una diferencia significativa, dando mayor ventaja a programar en C, con la excepción de operaciones que necesiten optimizarse de manera muy concreta. Además, el uso de temporizadores e interrupciones permiten llevar un mejor control sobre el flujo del programa, evitando *locks* que se pueden ocasionar por funciones como *delay*. -Miguel González Borja

Las diferencias entre el código auto-generado por el compilador de C y los programas escritos en ensamblador fue menor a la esperada. Posiblemente se debe a que los programas eran de muy baja complejidad y por ende no muy difíciles de optimizar. Esto se presta al desarrollo de programas más complejos que puedan tener varias alternativas de implementación a nivel ensamblador y en C para observar si las diferencias aumentan. Las interrupciones demostraron ser útiles en el manejo tanto de eventos externos al microcontrolador como en el caso de los fotodiodos como de eventos internos como los temporizadores. -Luis Felipe Landa Lizarralde

Utilizar el lenguaje ensamblador es muy útil para poder entender mejor los procesos que se realizan al momento de la ejecución de un código. Esto permite crear programas más eficientes y hacer rutinas que no se podrían en lenguajes de alto nivel. En general, los programas en ensamblador son más eficientes; sin embargo, en este caso, como los algoritmos implementados no eran complejos, no hubo una gran diferencia entre el código ensamblador y el auto-generado por C. Lo único que se dificultó un poco más al utilizar el lenguaje ensamblador es que el ATMega2560 no cuenta con instrucción para dividir, por lo que se tuvo que implementar un algoritmo para realizar división entera. Por otro lado, los temporizadores y las interrupciones son una gran herramienta para mejorar el flujo del programa. De esta manera, no necesitamos tener muchas condiciones en nuestro código para checar si ya ocurrió algún evento y además reducimos el riesgo de perder información cuando se utilizan *delays*. -Andrea Marín Alarcón

6. ROL

Miguel González Borja: Armado de circuitos de los sistemas, programación de sistemas 4.2.1 y 4.2.2. Sección de Resultados y Desarrollo.

Luis Felipe Landa Lizarralde: Implementación de programa de cálculo de discriminante, promedio y sistema 4.3.3. Sección Abstract, Introducción y Desarrollo.

Andrea Marín Alarcón: Programación de interrupciones y temporizadores. Marco Teórico y Desarrollo.

7. FUENTES CONSULTADAS

[1] A. Michael Wood, "Assembly language: How to learn to code assembly today," accessed 2019-02-12. [Online]. Available: https://www.ecured.cu/Lenguaje_ensamblador

[2] C. Hope, "Assembly language," accessed 2019-02-20. [Online]. Available: https://www.computerhope.com/jargon/a/al.htm

[3] EcuRed, "Lenguaje ensamblador," accessed 2019-02-20. [Online]. Available: https://www.ecured.cu/Lenguaje_ensamblador

[4] N. Hack, "Microcontroller - a beginners guide - basic and default usage of a timer and counter and the microcontroller clock," accessed 2019-02-20. [Online]. Available: https://www.newbiehack.com/TimersandCountersDefaultandBasicUsage.aspx

[5] A. Beginners, "Timers," accessed 2019-02-20. [Online]. Available: http://www.avrbeginners.net/architecture/timers/timers.html

[6] Atmel, "Atmel atmega640/v-1280/v-1281/v-2560/v-2561/v datasheet." [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf

[7] Mayank, "Introduction to avr timers," accessed 2019-02-20. [Online]. Available: https://maxembedded.com/2011/06/introduction-to-avr-timers/

[8] E. Foru, "Ir led — infrared led — infrared sensor," accessed 2019-02-20. [Online]. Available: https://electronicsforu.com/resources/learn-electronics/ir-led-infrared-sensor-basics

[9] Electronics—Projects—Focus, "Photodiode working principle, characteristics and applications," accessed 2019-02-20. [Online]. Available: https://www.elprocus.com/photodiode-working-principle-applications

Pin	I/O	Elemento
13	Output	LED Rojo
12	Output	LED Amarillo
11	Output	LED Verde

Cuadro 5
Configuración de pines para el sistema 4.3.3

Sistema	Preescalamiento	Modo	TCNT1/OCR1A	Interrupcion
4.3.1	1024	Normal	0xE0C0	TOIE1
4.3.2	1024	CTC	0x0FA0	OCIE1A
4.3.3	1024	CTC	0x3E80	OCIE1A

Cuadro 6
Configuración de temporizador 1 para los sistemas 4.3.1, 4.3.2 y 4.3.3

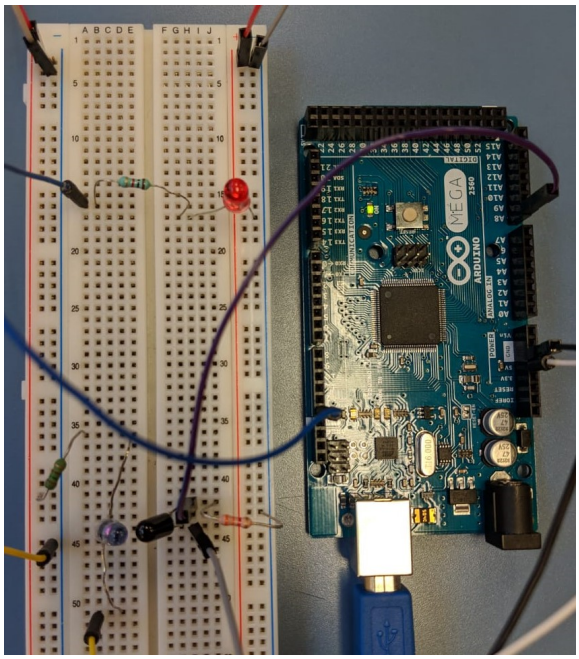

Figura 1. Sistema 4.2.2

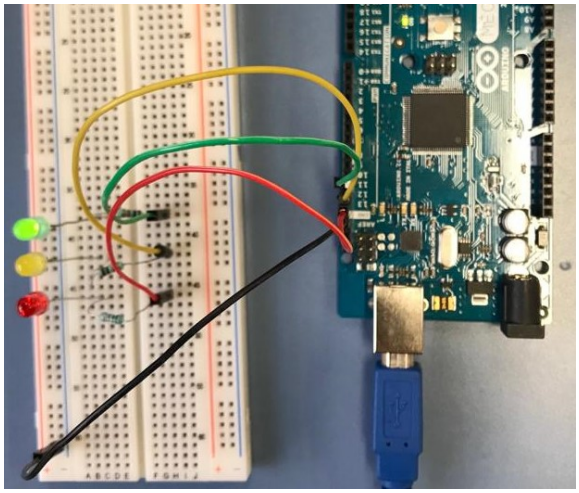

Figura 2. Sistema 4.3.3