Programación de Sistemas Embebidos 2020
Facultad de Informática
Universidad Nacional del Comahue
Trabajo Práctico 5 v0.1

Objetivo general: En este trabajo práctico utilizaremos el timer0 de 8 bits para componer diferentes aplicaciones:

- Para parpadear un led con periódos precisos
- Para desarrollar un real-time clock
- Para evaluar el rendimiento de la E/S serial
- Para ejecutar tareas periódicas utilizando timers por software
- Para construir una placa de sonido externa de 8bits 11khz

Timers/Contadores - Interrupciones

El timer/contador timer0 es un timer de 8bits. El MCU atmega328p contiene 3 timers, dos de 8 bits y uno de 16 bits. Los 3 son similares, y soportan una gran cantidad de modos. Para utilizar el timer0 simplemente debe configurarse el modo de trabajo, y "encender" el dispositivo (el dispositivo se enciende al configurar el prescalar u origen externo del reloj). Si los tres bits del prescalar se ponen a cero (0:0:0) se detiene ("apaga") el dispositivo (el timer/contador, es decir, deja de contar).

Los timers simplemente "cuentan" eventos. Los eventos pueden ser pulsos del reloj del sistema, o pulsos externos. Los timers pueden configurarse para generar una interrupción cuando el "conteo" llega a algún valor definido por el programador, lo que permite definir interrupciones precisas con respecto al reloj del sistema o eventos externos. Esto posibilita realizar acciones en momentos exactos disparados por acciones del hardware.

Además, los timers suelen presentar señales de salida, que son ondas cuadradas pero con ciclo de trabajo activados por el hardware (pero definidos por el programador), lo que permite simular una señal analógica, o definir cargas de trabajo útiles para control de motores, servos, y otros dispositivos que requieran señales con diferentes tensiones o periódos.

En este TP también comenzamos a utilizar avr-gcc junto a la biblioteca avr-libc. En particular, utilizaremos la macro ISR (vector) y las funciones sei(); y cli();. Para poder utilizar estas funciones se debe incluir interrupt.h:

```
#define <avr/interrupt.h>
```

ISR(VECTOR) es una macro, que se traduce a varios detalles de bajo nivel para colocar una rutina de atención de interrupciones. Es decir, que el compilador (junto con el preprocesador), al traducir la macro ISR() realiza lo siguiente:
- agrega una función de manejo de interrupciones para VECTOR, modificando la tabla de vectores e introduciendo el vector (salto a la rutina de atención de interrupciones) implementada en ISR() { }. Las interrupciones globalmente están deshabilitadas.
- como las sentecencias en C dentro de la ISR utilizan registros del procesador, la macro y el compilador, automaticamente agrega las instrucciones necesarias para salvaguardar el contexto del procesador, y restaurarlo al final de la ISR.
- agrega al final la instrucción máquina AVR reti, que vuelve de la ISR, y retorna la ejecución al programa, a la instrucción donde se interrumpió.

sei(); set the global interrupt bit. Habilita las interrupciones globalmente. A partir de este momento, todos los periféricos configurados para generar interrupciones, comenzarán a ser atendidos por el procesador cuando produzcan interrupciones.

cli(); unset the global interrupt bit. Deshabilita las interrupcioens globalmente.

NOTA: tenga en cuenta las posibles secciones críticas si se comparten variables globales o flags, entre bloques de código de la aplicación y las ISR, ya que podrían ser ejecutados concurrentemente.

Programación de Sistemas Embebidos 2020
Facultad de Informática
Universidad Nacional del Comahue
Trabajo Práctico 5 v0.1

Ejercicio 1. Real-time toggle led.

Utilizando el borrador de driver de las transparencias (el cual contiene ya una rutina de atención de interrupciones, que se ejecuta una vez por milisegundo) implementar el parpadeo de un led cada un segundo (un segundo apagado, un segundo encendido, etc).

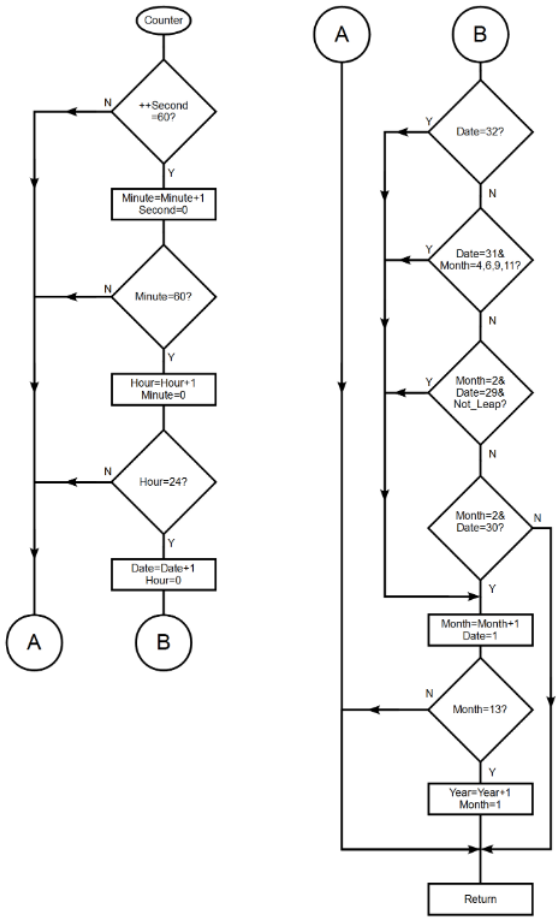
Comprenda la implementación del driver, y COMO el driver configura el dispositivo para generar una interrupción cada un milisegundo: observe que se usa el modo CTC (reset cuando la comparcación contra OCRA coincide).

Una implementación posible es esperar a que la variable “ticks” sea igual a 0 en el while infinito de main(). Entonces, hacer un “toggle” del led (la ISR pone a cero ticks cuando llega a 1000 milisegundos transcurridos). Esta idea estaría bien, pero analice las transparencias de clase para observar por qué “no funcionaría” implementado de manera directa. Por lo que este diseño podría necesitar un reajuste. Esboce un ejemplo y analice la situación.

Ejercicio 2. Implementar usando el timer0 un real-time clock.

Según “AVR134: Real Time Clock (RTC) Using theAsynchronous Timer APPLICATION NOTE”, un real time clock sigue el siguiente algoritmo:

Figure 4-1. Flow Chart, TCC0 Overflow ISR



Es decir, se debe mantener simplemente la hora:minuto:segundo, y el dia:mes:año. Debido a que el timer0 posibilita muchisima mas resolución (milisegundos o microsegundos transcurridos), se sugiere mantener también las décimas de segundo que van transcurriendo.

El valor del real-time clock debe ser mostrado al usuario en la PC a través del UART (utilice el driver serial ya implementado). Una salida posible en cutecom podría ser:

20:52:35.7

(el cual indica 20hs, 52min, 35seg, y 7 décimas de segundo).

Una posible implementación es como sigue:

1- utilice el driver timer.c de ejemplo provisto en las transparencias. El mismo está en modo CTC configurado para generar una interrupción cada vez que transcurrió un milisegundo (el timer "cuenta" en el registro tcnt desde 0 hasta 255, luego recomienza en cero. Cuando es cero genera una interrupción).

En la rutina de atención de interrupciones, se incrementa la variable ticks. Cuando la variable ticks es igual a 1000, significa que transcurrió un segundo, por lo que se debería agregar a la rutina de atención de interrupciones el algoritmo del diagrama de flujo, para mantener las variables o estructura que mantengan la fecha y hora del real-time clock. También reiniciar ticks a cero cuando alcanza el valor 1000.

2- En main() envíe simplemente, en un bucle infinito, la hora:minuto:segundos.décimas (de segundo). Una posible implementación es como sigue :

```
char buf[12];
for (;;) {
          timer0_rtc_to_str(buf);      /* devuelve la información del real-time clock en un string char[] */
          serial_put_char('\r');       /* esto ubicará el cursor en la primer columna de la línea actual,
                                        * por lo que el proximo serial_put_str() sobreescribirá la hora anterior
                                        */
          serial_put_str(buf);

          d = timer0_rtc_get_tos();             /* devuelve del rtc la décima de segundo actual (tenth of a second) */
          while (d == timer0_rtc_get_tos());    /* esperamos a que cambie la décima de segundo */

}
```

3- agregue la función timer_rtc_str() y timer0_rtc_get_tos(); al driver timer.c

El objetivo es lograr ver el real-time clock en acción en la PC (en lo posible, con décimas de segundo).

Opcional: alguna manera de establecer la fecha y hora actual del real-time clock, enviando por ejemplo comandos desde cutecom desde la PC.

Ejercicio 3. Evaluación de rendimiento para la E/S serial.

Realice un mini programa para contabilizar la cantidad de E/S serial realizada por segundo. Presente al usuario a través de la conexión serial los resultados. Ejemplo de implementación:

Agregue al real-time clock las funciones de

```
/* define la fecha y hora del rtc */
timer0_rtc_set(int year, int month, int day, int hour, int min, int sec);


/* devuelve el valor de los segundos del rtc */
timer0_rtc_get_sec();
```

Posible implementación en main:

```
cant_serial = 0;
timer0_rtc_set(0, 0, 0, 0, 0, 0);
sei();  /* habilita las interrupciones, enciende el rtc */

while (timer0_rtc_get_sec() == 0) {
    v = serial_put_char('A');
    cant_serial++;
}

/* enviar el resultado (cant_serial) al usuario */

for(;;);   /* un programa embebido nunca debe finalizar */
```

Compruebe que la contabilidad realizada en real-time coincide con la velocidad de transmición teórica/datasheet del uart. Pruebe este ejercicio con una velocidad alta del serial (por ejemplo: a 115200bps)

Ejercicio 4. Software Timers. Tareas periódicas.

Una función útil de un timer por hardware es implementar varios timers por software que utilicen el mismo timer hardware. Una implementación sencillas de software timers es simplemente usando contadores por software. Se los inicializa con un valor, y se los actualiza (decrementa) en cada tick de un timer de hardware (por ejemplo en la ISR). Cuando uno de los timers de software llega a cero, se ejecuta la tarea periódica asignada al timer de software.

Los timers de software pueden ser implementados facilmente como una estructura global compartida, entre la rutina de atención de interrupciones del timer de hardware (que mantendrá el estado de todos los timers de software), y la aplicación embebida. O como una biblioteca aparte que encapsule su funcionamiento.

Utilice en este ejercicio la implementación sencilla de software timers que se encuentra en la biblioteca tasks.c. Analice su código y observe cómo main() define un timer por software para una tarea que parpadea un led. Compilar y verificar/testear.

Este ejercicio presenta dos conceptos:

- comprender los timers por software; y
- comprender el CONCEPTO de que las funciones en lenguaje C tienen una dirección, y por lo tanto, pueden ser utilizadas como punteros.

4.a. Describa con uno o dos párrafos cómo funciona la biblioteca sencilla que implementa timers por software.

4.b. Utilice la biblioteca para definir y ejecutar 4 tareas periódicas (por ejemplo, una cada 100ms, otra cada 200ms, otra cada 500ms y otra cada 1seg). En cada tarea realice un único trabajo, por ejemplo, en una tarea parpadee un led, en otra parpadee otro led, en otra envíe la hora por el serial, etc.

Ejercicio 5. Configurar el hardware, y desarrollar el software de una placa de sonido 8bits. Señales PWM.

El objetivo de este ejercicio es controlar señales PWM generadas por el timer, para construir y emitir una señal de salida a ser convertida en audio por un parlante mono (ejemplo: cualquier parlante o auricular con ficha jack audio).

Una señal de audio analógica es una señal variante en el tiempo continua, con tensiones que no superan los 1.7v o 2.0v En algunas ocasiones, dependiendo de diferente factores (ej. Db) estas tensiones de referencia pueden variar.

Una forma sencilla de generar una señal de audio es emitiendo una señal PWM a través de un filtro RC (un circuito RC es un circuito compuesto por una resistencia y un capacitor).

Programación de Sistemas Embebidos 2020
Facultad de Informática
Universidad Nacional del Comahue
Trabajo Práctico 5 v0.1

Las señales PWM que pueden generar los timers de los AVR permiten variar la tensión de salida en un rango de 0V a 5V. Por ejemplo, si durante un período la mitad del tiempo la señal está en alto y luego la mitad de tiempo en bajo, y si la frecuencia de este período es alta (miles de veces por segundo), la tensión de salida estará al 50%, lo que al ser medida dará un voltage de 2.5V (en un AVR trabajando con un VCC de 5V).
AL tiempo, del total del período, en que la señal está en nivel alto se lo llama **ciclo de trabajo (duty cycle).** Como acabamos de indicar, un ciclo de trabajo del 50% dará una tensión de salida de 2.5v. Un ciclo de trabajo del 20% (20% en alto, 80% en bajo) dará una tensión de salida de 1v. Etc. Siempre proporcional.
La señal PWM no es analógica (de hecho, es muy cuadrada), pero al variar el ciclo de trabajo en una alta frecuencia, y al colocar un filtro RC, permite suavizar levemente esta señal cuadrada de tensión variable, y generar una señal "algo similar" a una señal analógica de audio real. Un parlante o auricular que obtenga la señal generará una señal de audio con bastante ruido, pero entendible y correlacionado con el audio original.

Los timers AVR permiten ser configurados en modo Fast PWM, lo cual genera automaticamente la señal de salida en uno de sus pines de salida del timer.
El ciclo de trabajo es programable, y se utilizan los registros comparadores. Por ejemplo, usando el timer0, el programador podría configurarlo de la siguiente manera:

- que el timer/counter cuente pulsos del reloj del sistema (con prescalar)
- modo fast pwm
- que el timer/counter al llegar al tope (0xFF) vuelva a comenzar a contar de cero (0x00)
- que la señal PWM de salida se ponga en alto automaticamente cuando el timer/counter == 0 (igual a cero)
- que la señal de PWM de salida se ponga en bajo automaticamente cuando el timer/counter sea igual al registro comparador

Entonces, el programador puede escribir en el registro comparador los valores que desee, para generar el ciclo de trabajo (y por lo tanto una tensión de salida de la señal PWM) exacto que se necesite.

Ejemplo: con la configuración anterior, si el timer/counter va de 0x00 a 0xFF (cuenta) y el programador coloca en el registro comparador el valor 50 (aproximadamente el 20% de 0xFF), la tensión de salida será del 20% de 5v, es decir, de 1v.

Teniendo lo anterior presente, ahora comentaremos sobre la señal de audio.
Una señal de audio digitalizada es representada por valores discretos. Es decir, que cuando se digitalizó, se tomaron valores de la señal de audio analógica, en una frecuencia. En nuestro caso, si la señal de audio a reproducir es de 8bits 11khz, significa que se tomaron 11000 muestras por segundo (11khz), y de cada muestra se obtuvo un valor de 8bits que representa a la señal original (es decir, que la resolucion en 8bits es de 256 valores). Si bien esta señal es de muy baja calidad (se la llama comunmente "de radio", porque es similar a la obtenida y escuchada en una radio AM), puede ser entendible claramente, sobre todo si el audio representa voces humanas ("personas hablando").

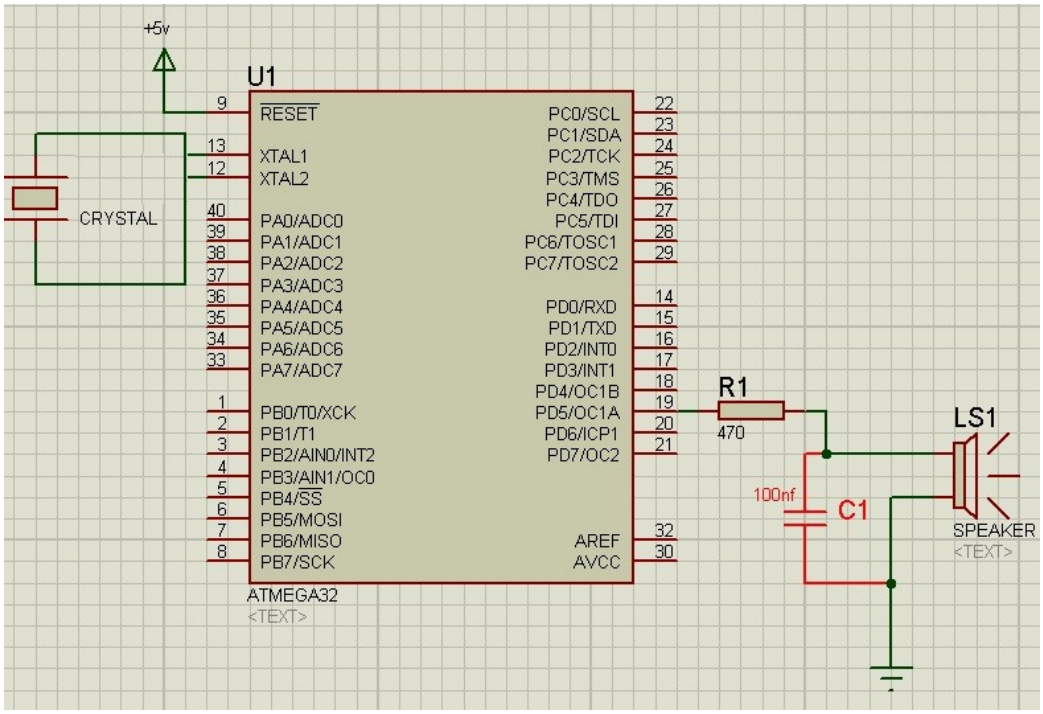
https://es.wikipedia.org/wiki/Audio_digital

http://aprendeenlinea.udea.edu.co/boa/contenidos.php/11dc9853df8fe7a0e6ea5b13554f1866/361/1/contenido/sonido.html

Lo que vamos a realizar en este ejercicio es reproducir algunos archivos de audio en nuestra PC, a 8bits 11khz. Utilizaremos un script en la PC que transmita los bytes de uno de estos archivos de audio a reproducir por la "placa de sonido AVR". Transmitiremos los bytes por el serial, a una velocidad aproximada de 11khz (por ejemplo, a 115200bps).

Programación de Sistemas Embebidos 2020
Facultad de Informática
Universidad Nacional del Comahue
Trabajo Práctico 5 v0.1

Hardware



(La resistencia R1 y el capacitor C1 conforman un low-pass filter. El jack de audio será conectado en LS1.)

Software

La aplicación embebida del AVR deberá ir obteniendo los bytes del audio digital desde la PC, a 11khz aproximadamente (si es exacto mejor), y convertir el valor del byte en una señal PWM acorde. Ejemplo: si el byte que llegó desde la PC tiene el valor 10, y como el valor 10 (0x0A) es el 10% de 255 (0xFF) (255 es el valor máximo del audio digital en 8bits), se deberá emitir una señal PWM que genere el 10% de 1.7v (o el 10% de 2v).

Una posible implementación es usando el timer1, ya que posibilita 16bits. Si un audio digital a 8bits tiene valores de 0 a 255, y si queremos que 255 ya represente el tope de tensión de la señal de audio de salida PWM (aproximadamente 1.7v o 2v), podemos usar entonces el timer1, e indicar que el tope del timer/counter es 1000. De esta manera, se podría utilizar como ciclo de trabajo el valor digital del audio en 8bits que proviene de la PC; donde el máximo valor en 8bits, 255, si usado como valor comparador del timer, tiene que corresponder a una señal de salida del PWM de aproximadamente 1.7v de tensión.

Si la idea anterior le parece válida, entonces aquí hay mas detalles posibles:

Recibimos desde la PC el wav de 8bits a 11khz por el serial UART y lo usamos para generar el ciclo de trabajo de la señal PWM. Para esto:
- Activamos el timer1 con una frecuencia de 11000 períodos por segundos.
- En cada perídodo contamos de 0 a 1000. Utilizanco el registro ICR1 como tope. El valor 1000 es estimativo, ya que el valor máximo de audio a 8bits es 255, y debería generar unos 2v de salida usando PWM. Como 255 es aproximadamente un 25% de 1000; se podría utilizar el byte de audio 8bits directamente como comparador del ciclo de trabajo del PWM (al llegar al valor de este comparador la señal del PWM baja automaticamente, y contínua en bajo hasta que el timer1/coounter llega al tope 1000. (como 255 es aproximadamente el 25% de 1000, el ciclo de trabajo tendría un máximo del 25% de 5V, que es aproximadamente una tensión de 1.7v en la señal de salida del pwm).
- Activamos Fast PWM y utilizamos el registro OCR1A para generar el voltage de 0 a 1.7v/2v que necesita la salida de audio (el audio jack).

- El audio jack está conectado al pin de salida OC1A.
- Activamos las interrupciones del timer1 en overflow. Es decir, 11khz (11000) interrupciones por segundo.

- En el bucle infinito de main iremos obteniendo cada byte del audio digital a 8bits desde la PC (los bytes tienen que arribar a 11khz, por lo tanto el serial debería estar trabajando a 115200bps), y debe ser colocado en alguna variable compartida con el timer para sincronismo.
- En la rutina de atencion de interrupciones colocamos el byte que esté en el buffer compartido para sincronismo con main en OCR1A, porque es el próximo byte de sonido a generar un ciclo de trabajo en el PWM. Como hay 11000 interrupciones por segundo los bytes recibidos por UART son consumidos por la rutina de interrupciones casi sin pérdidas de bytes recibidos.

Luego, una vez que podamos reproducir archivos de audio particulares, se puede combinar ffmpeg para que tome el audio del sistema Linux, lo convierta en "tiempo-real" a 8bits/11khz, y se envíe por el serial, para que la "placa de sonido AVR" lo reproduzca. Esto hará que el sistema de cómputo tenga una placa de sonido externa rudimentaria, pero creada sólo con un timer y señal PWM.
Nota de color: Steve Wozniak utilizó una señal PWM a través de un pin que quedaba sin uso del decodificador de memoria de la Apple II, para generar audio en esa computadora usando esta técnica.

Ejecución y verificación:

Para convertir en la PC un mp3 a 8bits, 11khz, se puede usar el siguiente comando ffmpeg:

```
# el archivo de entrada puede estar en otro formato, no sólo en mp3
ffmpeg  -i archivo_input.mp3 -ar 11000 -acodec pcm_u8 -ac 1  archivo_8bit.wav
```

Una vez que el programa está en funcionamiento en el avr debemos transferir el wav.
En Linux podemos configurar el serial stty y usar cat. Ejemplo:

```
stty -F /dev/ttyUSB0   # nos muestra la configuracion
stty -F /dev/ttyUSB00 speed 115200  # configuramos el driver serial del Linux a 115200
stty -F /dev/ttyUSB0   # revisamos la nueva configuracion

cat archivo_8bit.wav > /dev/ttyUSB00      # enviamos el archivo wav al avr
```

Verificación: compilar, vincular y enviar el firmware AVR cada ejercicio. Verificar el funcionamiento de la aplicación de manera progresiva. Grabar un pequeño video de unos pocos segundos explicando la tarea.

Entrega: subir (push) el trabajo práctico resuelto (o sus versiones intermedias) al repositorio git compartido (http://github.com/zrafa/pse2020.git)