LAB 3: EXPLICACION

1. Configuración Inicial

El programa empieza con algunas configuraciones:

```
.ORG 0x0000
jmp start
```

- `.ORG 0x0000`: Esta directiva coloca el código en la dirección de memoria 0x0000.

```
.ORG 0x0008          ; Dir de memoria de la interrupcion de bo
jmp     pcint ;
```

Un "dir de memoria de la interrupción de botones" es la **dirección de memoria** donde se encuentra el vector de interrupción asociado a la pulsación de un botón en un sistema, lo que le dice al procesador qué hacer cuando el botón se presiona.

- `jmp start`: Salta inmediatamente al inicio del programa principal.

```
.ORG 0x0008
jmp pcint
.ORG 0x001C
jmp _tmr0_int
```

- `.ORG 0x0008` y `.ORG 0x001C`: Configura las direcciones de interrupción para el manejo de botones (pcint) y el temporizador (tmr0), que son gestionadas más adelante en el código.

2. Configuración de los puertos

```
ldi r16, 0b00111101
out DDRB, r16
out PORTB, r16
```

- `DDRB` se configura para que los pines PB2 a PB5 sean salidas (para controlar LEDs), y PB0 sea la salida de datos seriales para el display.
- `PORTB` se inicializa para apagar los LEDs conectados a esos pines.
- ldi `0b00111101`: Es un valor en binario que configura los pines del puerto B como entradas o salidas.

```
ldi     r16,    0b00000000
out     DDRC,   r16         ; 3 botones del shield son entrad
```

- carga el registro
`R16` con `0b00000000` y luego lo envía al registro **DDRC** para configurar **todos los pines del puerto C** como **entradas**. Esto es necesario porque el microcontrolador necesita **leer** el estado de los botones conectados al puerto C, y para poder hacer lecturas, esos pines deben estar configurados como entradas.

¿QUE ES UN LATCH?

Es un mecanismo que guarda temporalmente los datos antes de enviarlos a la salida (en este caso, el registro paralelo del display). En este código, los pines PD4 y PD7 del microcontrolador se usan para controlar el latch.

```
ldi r16, 0b00000000
out DDRC, r16
```

- `DDRC` se configura para que los pines del puerto C sean entradas (probablemente para botones).

```
ldi r16, 0b10010000
out DDRD, r16
```

- `DDRD` se configura para que los pines PD4 y PD7 sean salidas (para controlar el reloj y el latch del display de 7 segmentos).

```
cbi PORTD, 7
cbi PORTD, 4
```

- Aquí, los pines PD7 y PD4 se ponen en bajo (0). Esto inicializa el reloj y el latch en su estado de reposo.
- `cbi PORTD, 7`: Baja el pin PD7 a 0, que es el reloj serial.
- `cbi PORTD, 4`: Baja el pin PD4, que controla el latch, para indicar que no se transfiera información aún.

DISPLAY DE 7 SEGMENTOS:

- Es un dispositivo con 7 LEDs dispuestos en forma de número que puede mostrar caracteres del 0 al 9. Cada segmento es controlado por un bit en el registro. En el ejemplo, se utilizan registros para indicar qué segmentos deben encenderse.

Resumiendo:

Este bloque de código configura los pines del microcontrolador para controlar el **reloj serial** y el **reloj del latch**. Ambos relojes son necesarios para manejar la comunicación y el control del **display de 7 segmentos** que será utilizado en el programa. Estos relojes permiten que los datos se transmitan correctamente y se sincronicen con el display para mostrar la información en el formato correcto.

- **PD.7** → Reloj serial (para enviar datos en serie).los datos seriales en tu código son los bits de información que se envían uno por uno, sincronizados con un reloj, para comunicar información entre dispositivos.
- **PD.4** → Reloj del latch (para sincronizar cuándo se deben mostrar los datos).

3. Configuración de botones

```
ldi r19, 0b000001110
sts PCMSK1, r19
ldi r17, 0b00000010
sts PCICR, r17
```

- `PCMSK1`: Habilita las interrupciones por cambio de estado para tres botones conectados al puerto C.
- `PCICR`: Activa las interrupciones para los botones.
- `ldi r19, 0b000001110` y `sts PCMSK1, r19`: Configura las interrupciones por cambio de estado en los pines de los botones conectados al puerto C. Aquí se habilitan interrupciones en los pines PC1, PC2 y PC3.
- `ldi r17, 0b00000010` y `sts PCICR, r17`: Habilita las interrupciones de cambio de pin (pin change interrupt) en el grupo correspondiente a los botones.
- `0b00000010` se usa porque habilita las interrupciones por cambio de pin para el **puerto C**.
- Los **botones** en este programa están conectados a **PC0**, **PC1**, y **PC2**, por lo que necesitamos habilitar las interrupciones solo para este puerto.
- **PCICR**: Habilita las interrupciones por cambio de pin para **todo un puerto** (**PortB**, **PortC**, o **PortD**).
  - Ejemplo: En el código que mencionas, se habilitan las interrupciones para **PortC** usando `PCICR, 0b00000010`.
- **PCMSK1**: Selecciona los **pines específicos** dentro del **PortC** que pueden generar una interrupción por cambio de pin.
  - Ejemplo: Con `PCMSK1, 0b00000111`, se activan las interrupciones solo para los pines **PC0**, **PC1**, y **PC2** del **PortC**.

`PCMSK1` (máscara de pines):

- Este registro define **qué pines específicos** dentro de un puerto generan interrupciones.
- Cada bit de `PCMSK1` corresponde a un pin individual de un puerto (por ejemplo, el puerto C en el caso de `PCMSK1` en algunos AVR). Si un bit está en `1`, entonces se habilita la interrupción para ese pin; si está en `0`, no se habilita.

- **Ejemplo:**
  - Si configuras `PCMSK1 = 0b00000111`, las interrupciones se habilitarán para los **pines 0, 1, y 2** del puerto asociado a `PCMSK1`. Estos pines ahora están listos para detectar cambios de estado y generar una interrupción.

2. `PCICR` (control de interrupción por cambio de pin):

- Este registro controla **qué grupo de pines** (o qué **puerto completo**) puede generar interrupciones por cambio de estado.
- En los microcontroladores AVR, los pines están organizados en **grupos**. Por ejemplo, un grupo podría ser el puerto B (`PCMSK0`), otro grupo podría ser el puerto C (`PCMSK1`), y otro el puerto D (`PCMSK2`).
- Cada bit de `PCICR` habilita o deshabilita las interrupciones por cambio de estado para un grupo de pines específico.
- **Ejemplo:**
  - Si cargas `PCICR = 0b00000010`, solo habilitas las interrupciones para el **grupo 1** de pines, que corresponde a los pines controlados por `PCMSK1`.

Analogía simple:

- `PCMSK1` es como una lista de verificación que te permite decidir qué pines (botones) quieres monitorear para detectar cambios.
- `PCICR` es como un interruptor general que enciende o apaga la funcionalidad de interrupciones para todo un grupo de pines (por ejemplo, todos los pines del puerto C).Sin esta habilitación, aunque los pines estén configurados correctamente, no se generaría la interrupción.

Resumen:

- `PCMSK1` habilita interrupciones **en pines específicos** del grupo correspondiente (por ejemplo, los pines del puerto C).
- `PCICR` habilita las interrupciones **para todo un grupo de pines o un puerto** (por ejemplo, todos los pines gestionados por `PCMSK1`).

4. Configuración del temporizador

```
ldi r16, 0b00000010
out TCCR0A, r16
```

```
ldi r16, 0b00000101
out TCCR0B, r16
ldi r16, 62
out OCR0A, r16
ldi r16, 0b00000010
sts TIMSK0, r16
```

- `TCCR0A` y `TCCR0B`: Configuran el Timer 0 para contar hasta 62 (que se compara con `OCR0A`) y genera una interrupción cuando ocurre la coincidencia.
- `TIMSK0`: Habilita la interrupción del Timer 0.
- `ldi r16, 0b00000010`, `out TCCR0A, r16`: Configura el Timer0 para contar hasta el valor en `OCR0A` y luego reiniciar (modo CTC).
- `ldi r16, 0b00000101`, `out TCCR0B, r16`: Configura el prescaler del temporizador a 1024 (divide la frecuencia del reloj por 1024).
- `ldi r16, 62`, `out OCR0A, r16`: Establece el valor de comparación en 62, lo que determina cada cuánto tiempo ocurre la interrupción del Timer0.

Bits importantes de `TCCR0A` en el modo CTC:

- **Bit** `WGM01 = 1`: Selecciona el modo **CTC**.
- **Bit** `WGM00 = 0`: Esto es parte de la selección del modo de operación, y en conjunto con `WGM01` selecciona CTC.

Así que, para habilitar el **modo CTC**, necesitas:

- `WGM01 = 1` (bit 1 en `TCCR0A`).
- `WGM00 = 0` (bit 0 en `TCCR0A`).

El valor binario `0b00000010` corresponde a:

Bit	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	1	0

Esto significa que `WGM01 = 1` y `WGM00 = 0`, lo que efectivamente selecciona el **modo CTC**.

¿Por qué 62?

- **El temporizador genera interrupciones más rápidas** (cada ~3.97 milisegundos en tu caso).
- Estas interrupciones rápidas son acumuladas por un contador (`r25` en tu código).
- Cuando el contador ha acumulado suficientes interrupciones (250 interrupciones), genera un evento a **1 Hz** (es decir, cada segundo).

En tu código, configuras el temporizador **Timer0** para generar una interrupción periódica:

```
ldi r16, 62          ; Valor de comparación
out OCR0A, r16       ; Establece 62 en OCR0A
ldi r16, 0b00000101  ; Prescaler = 1024
out TCCR0B, r16      ; Configura el temporizador con presca
ler de 1024
```

- Esto genera una interrupción **cada 63 ciclos** del temporizador (porque cuentas de 0 a 62, lo que da 63 ciclos).
- Como el prescaler está en 1024, la frecuencia efectiva del temporizador se reduce a:

$$\frac{16,000,000}{1024} = 15,625 \text{ Hz}$$

- Con un valor de **OCR0A = 62**, el temporizador genera una interrupción cada:

$$\frac{63}{15,625} = 0.004032 \text{ segundos} \approx 4.03 \text{ milisegundos}$$

En el código, después de que el temporizador genera cada interrupción (cada 4.03 milisegundos), usas un contador adicional (`r25`) para contar cuántas interrupciones han ocurrido hasta sumar **1 segundo**. El cálculo es el siguiente:

- **1 segundo** = **1000 milisegundos**.
- Cada interrupción ocurre cada **4.03 milisegundos**.
- Para llegar a **1 segundo** necesitas:

$$\frac{1000}{4.03} \approx 248 interrupciones$$

Sin embargo, en tu código, parece que has redondeado este valor a **250** interrupciones, lo cual es razonable y suficientemente cercano a 1 segundo.

```
sts     TIMSK0, r16         ;habilito la interrupcion (falta
```

Este código está **habilitando la interrupción de comparación A** (OCIE0A) del Timer 0. En el modo CTC que has configurado, cuando el Timer 0 cuente desde 0 hasta el valor en OCR0A (que es 62), el temporizador genera una interrupción y ejecutará la rutina de interrupción correspondiente.

5. Inicialización de registros

```
ldi     r29, 0b00000100
    ldi     r27, 0  ; registro auxiliar
    ldi     r25, 250    ; contador para generar un segundo
    ldi     r24, 0x00       ;inicializo r24 para un contado
r generico
    ldi     r23, 0 ; act display
    ldi     r22, 0
    ldi     r21, 0
    ldi     r20, 0
    ldi     r19, 0
    .def    estado_botones=r19
    .def    contador_decenas=r20
    .def    contador_centenas=r21
    .def    contador_millares=r22
    .def    valor_display=r23
    .def    contador_generico=r24
    .def    contador_segundos=r25
    .def    registro_auxiliar=r27
```

- `r24`: Se inicializa como un contador genérico.
- `r25`: Se configura para contar 250 ciclos (esto representa 1 segundo).
-

En este programa, el registro `r27` es utilizado como un **registro auxiliar** para el manejo de datos relacionados con los botones y el display de 7 segmentos. Al comienzo del código, se inicializa con el valor `0`:

```
ldi r27, 0 ; registro auxiliar
```

A lo largo del código, `r27` se define como `registro_auxiliar` y se emplea en diversas rutinas, como por ejemplo para mover el valor de otros registros (como `estado_botones` o contadores) y realizar comparaciones o conversiones para traducir los valores que se deben mostrar en el display de 7 segmentos.

En estas rutinas, el registro `r27` juega un papel importante para almacenar temporalmente datos, comparar valores y realizar operaciones de cálculo. Por ejemplo, en la rutina `unidades` se usa para verificar si se ha alcanzado un desborde de unidades (para incrementar las decenas):

```
unidades:
    mov registro_auxiliar, estado_botones
    cpi registro_auxiliar,10
    brne traductor
    call desborde_unidades
    ret
```

Aquí, el valor de `estado_botones` se copia en `r27` (`registro_auxiliar`) y luego se compara para saber si se ha alcanzado el valor 10. De ser así, se llama a la rutina `desborde_unidades`, que actualiza el contador de decenas.

En resumen, el registro `r27` se utiliza como un **almacenamiento temporal para facilitar el manejo de valores y cálculos** en las rutinas de control del display de 7 segmentos y otros componentes.

Registro `r19` (alias `estado_botones`)

El registro `r19` se define como `estado_botones` y almacena el **estado actual de los botones**. Este valor se utiliza para contar las pulsaciones o determinar el estado de los botones conectados al puerto.

Ejemplo:

```
.def estado_botones=r19
```

En la rutina de manejo de botones, el valor de `estado_botones` se actualiza en función de las interrupciones generadas por la pulsación de botones:

```
assembly
Copiar código
inc estado_botones     ; Incrementa el estado de los botone
s
```

Registro `r20` (alias `contador_decenas`)

Este registro almacena el **contador de decenas** para el display de 7 segmentos. Se incrementa cuando el valor de las unidades sobrepasa 9 (es decir, ocurre un "desborde" de unidades).

Ejemplo:

```
assembly
Copiar código
.def contador_decenas=r20
```

En la rutina `desborde_unidades`, cuando las unidades llegan a 10, se incrementa el contador de decenas:

```
assembly
Copiar código
inc contador_decenas
```

Registro `r21` (alias `contador_centenas`)

Almacena el **contador de centenas**, que se incrementa cuando el contador de decenas alcanza 6 (en un reloj basado en minutos, por ejemplo).

Ejemplo:

```
assembly
Copiar código
.def contador_centenas=r21
```

Cuando las decenas alcanzan el valor 6, se produce un "desborde" y se incrementan las centenas:

```
assembly
Copiar código
call desborde_decenas
inc contador_centenas
```

Registro `r22` (alias `contador_millares`)

Este registro almacena el **contador de millares**, utilizado para contabilizar los miles en el display de 7 segmentos.

Ejemplo:

```
assembly
Copiar código
.def contador_millares=r22
```

Cuando las centenas llegan a 10, se incrementa el valor de los millares:

```
assembly
Copiar código
call desborde_unidades_m
inc contador_millares
```

Registro `r23` (alias `valor_display`)

Este registro almacena el valor actual que debe mostrarse en el display de 7 segmentos. Sirve como un **contador del dígito activo** que se actualiza periódicamente.

Ejemplo:

```
assembly
Copiar código
.def valor_display=r23
```

Se utiliza en la interrupción del temporizador para ir cambiando de dígito en el display:

```
assembly
Copiar código
inc valor_display
call act_display
```

Registro `r24` (alias `contador_generico`)

Este registro actúa como un **contador genérico** que puede ser utilizado para cualquier propósito dentro del programa. En este caso, parece no estar explícitamente utilizado en la lógica principal.

Ejemplo:

```
.def contador_generico=r24
```

Registro `r25` (alias `contador_segundos`)

Este registro se utiliza como un **contador de segundos**. Su valor inicial es 250, lo que corresponde a un segundo en función de la configuración del temporizador.

Ejemplo:

```
assembly
Copiar código
.def contador_segundos=r25
ldi r25, 250           ; Inicializa el contador de segundos
```

Registro `r26`

Es utilizado en la interrupción de botones para **almacenar temporalmente el registro de estado del procesador (SREG)** y restaurarlo antes de salir de la interrupción. Esto es necesario para preservar el estado de los flags del procesador durante la interrupción.

Ejemplo:

```
assembly
Copiar código
in r26, SREG           ; Guarda el estado del registro de e
status
out SREG, r26          ; Restaura el estado al salir de la
interrupción
```

Registro `r29`

Este registro se utiliza en varias rutinas para almacenar valores temporales, como en la interrupción de botones para detectar cuál botón fue presionado.
El registro
`r29` sirve como un registro de **control** en tu código. Dependiendo del contexto, se usa para manejar el estado de los botones (en la interrupción de cambio de pines) y para decidir si se debe realizar alguna acción durante la interrupción de temporización (1 Hz) del **Timer 0**.

Ejemplo:

```
assembly
Copiar código
```

```
sbis PINC, 1
ldi r29, 0b00000010   ; Botón 1
```

Resumen de los registros:

- **r16**: Registro temporal para configuraciones y datos.
- **r17**: Posición del dígito en el display de 7 segmentos.
- **r19 (estado_botones)**: Estado actual de los botones.
- **r20 (contador_decenas)**: Contador de decenas para el display.
- **r21 (contador_centenas)**: Contador de centenas.
- **r22 (contador_millares)**: Contador de millares.
- **r23 (valor_display)**: Valor del dígito activo en el display.
- **r24 (contador_generico)**: Contador genérico.
- **r25 (contador_segundos)**: Contador para el temporizador de un segundo.
- **r26**: Registro auxiliar para almacenar y restaurar SREG.
- **r27 (registro_auxiliar)**: Registro auxiliar para cálculos y operaciones temporales.
- **r29**: Utilizado en interrupciones para detección de botones.

6. Habilitación de interrupciones globales

```
sei
```

- `sei`: Habilita las interrupciones globales.

7. Rutina `sacanum`

Esta rutina envía los datos seriales para el display de 7 segmentos:

```
sacanum:
    call    dato_serie
    mov     r16, r17
    call    dato_serie
```

```
    sbi     PORTD, 4
    cbi     PORTD, 4
    ret
```

Explicación de `sacanum`:

1. **Primera llamada a `dato_serie`**:
   - Envía el contenido de `r16` al display de 7 segmentos. El valor de `r16` contiene el patrón de bits para encender/apagar los **segmentos individuales** del display.
   - Los bits en `r16` corresponden a los segmentos (a, b, c, d, e, f, g) del display y determinan qué segmentos se encenderán para formar un número (por ejemplo, para mostrar el número `0`, se encenderían los segmentos que forman un "0").
2. `mov r16, r17`:
   - Después de enviar los bits de los LEDs, se mueve el valor de `r17` a `r16`. En este caso, `r17` **contiene el valor del dígito** que se está actualizando (por ejemplo, si estás trabajando con un display de 4 dígitos, `r17` selecciona cuál de esos dígitos se está actualizando, ya que el display comparte los mismos segmentos).
3. **Segunda llamada a `dato_serie`**:
   - Ahora envía el contenido de `r17` (que fue movido a `r16`) al display. `r17` contiene la información de **cuál dígito del display se está activando**, controlando en cuál de los cuatro dígitos se debe mostrar el número.
4. **Pulso en el latch (`sbi` y `cbi` en `PORTD, 4`)**:
   - `sbi PORTD, 4`: Genera un pulso en el pin **PD4** (que parece ser el **reloj del latch**) para asegurar que el valor enviado al display de 7 segmentos sea cargado y se muestre.
   - `cbi PORTD, 4`: Baja el reloj después de dar el pulso, completando el ciclo.

En resumen, `sacanum` envía dos bytes de información al display: uno para definir **qué segmentos se deben encender** y otro para definir **en qué dígito del display** se debe mostrar el número.

Ejemplo concreto:

Imagina que quieres mostrar el número `0` en el primer dígito del display de 4 dígitos.

- `r16` tiene el valor `0b00000011`:
  - Este patrón enciende los segmentos necesarios para formar el número `0` en el display.
- `r17` tiene el valor `0b00010000`:
  - Este patrón selecciona el **primer dígito** (de los 4 posibles) en el display.

Flujo de datos:

1. **Primero, envías `r16`**:
   - Esto envía el patrón para mostrar el `0` al display.
2. **Después, `mov r16, r17`**:
   - Ahora, `r16` contiene el valor de `r17`, que selecciona el **primer dígito** del display.
3. **Finalmente, envías nuevamente `r16` (que ahora es `r17`)**:
   - Esto asegura que el número `0` se muestre en el **primer dígito** del display.

Flujo detallado de `sacanum`:

1. **Primera llamada a `dato_serie`**:
   - Cuando entras a la rutina `sacanum`, el registro `r16` contiene el **patrón de bits** que controla qué segmentos del display se encienden. Este es el **número que deseas mostrar**.
   - Llamas a `dato_serie` para enviar esos bits serialmente al display.
   - **Esto envía el valor de `r16`** (los segmentos) al display.
2. **Copiar `r17` a `r16`**:
   - Después de enviar el patrón de segmentos, **copias el valor de `r17` a `r16`** usando la instrucción `mov r16, r17`.
   - `r17` **contiene el número del dígito** del display que se está actualizando (por ejemplo, el primer dígito, el segundo dígito, etc.).
   - Ahora, `r16` **tiene el valor de `r17`**, que selecciona el dígito.

3. **Segunda llamada a `dato_serie`**:
    - Llamas nuevamente a `dato_serie`, pero esta vez lo que se envía es el **contenido de `r16`**, que ahora es el valor de `r17` (es decir, cuál dígito del display se va a activar).
    - **Esto envía el dígito seleccionado** al display.
4. **Pulso en el latch**:
    - Una vez que has enviado tanto el **patrón de segmentos** como el **dígito** al display, generas un pulso en el pin `PD4` (latch) para que el display cargue y muestre los datos enviados.

Diagrama visual del flujo:

1. **Envío del patrón de segmentos**:
    - `r16` = Patrón de bits para los segmentos (ejemplo: `0b00000011` para el número "0").
    - Llamada a `dato_serie` → **Se envían los bits de `r16` (patrón de segmentos)**.
2. **Mover el dígito al que apuntarás**:
    - `mov r16, r17` → **Copia `r17` (dígito) a `r16`**.
    - `r17` = Dato del dígito que se activará (ejemplo: `0b00010000` para el primer dígito del display).
3. **Envío del dígito**:
    - Llamada a `dato_serie` → **Se envían los bits de `r16` (ahora el dígito)**.
4. **Latch**:
    - Generas un pulso con `sbi` y `cbi` en `PORTD, 4` para que el display **cargue los datos** enviados y se actualice visualmente.

Entonces, el flujo de `sacanum` sería así:

1. **Primero** envías **los datos de `r16`**, que representan **qué segmentos** se deben encender para formar un número.
2. **Luego** copias `r17` **a** `r16` para seleccionar **qué dígito** va a mostrar el número.

3. **Finalmente**, envías el **valor de** `r16` **(que ahora es** `r17` **)** para decirle al display en qué **dígito** debe aparecer el número.

8. Rutina `dato_serie`

Explicación de `dato_serie`:

1. **Cargar** `r18` **con** `8`:
   - Esto configura un contador de **8 iteraciones** en `r18` para asegurar que se envíen exactamente **8 bits** (un byte completo) a través del bus serial.
2. **Bucle** `loop_dato1`:
   - `cbi PORTD, 7`: Baja el reloj serial (`SCLK = 0`) para preparar el envío del bit.
   - `lsr r16`: El registro `r16` es rotado a la derecha, y el bit menos significativo (LSB) se mueve al **carry** (`C`).
   - la rotación a la derecha con `lsr r16` es necesaria para extraer y preparar cada bit de `r16` para su transmisión en serie, sincronizándolo con el reloj serial controlado por `cbi PORTD, 7`
   - `brcs loop_dato2`: Si el carry está en `1`, salta a `loop_dato2`, lo que indica que el bit a enviar es un `1`.
   - Si el carry es `0`, la instrucción `cbi PORTB, 0` pone en `0` el pin de datos seriales (`SD = 0`), para enviar un `0`.
3. **Enviar el bit a través del pin de datos seriales (SD)**:
   - Dependiendo del valor del carry, se envía un `1` o un `0` al pin de datos seriales.
   - `sbi PORTD, 7`: Luego sube el reloj serial (`SCLK = 1`) para **enviar** efectivamente el bit.
4. **Decrementar** `r18`:
   - El contador `r18` se decrementa para contar los bits enviados. Cuando se llega a `0`, se han enviado los 8 bits (un byte completo).
5. **Repetir el proceso**:
   - El bucle continúa enviando los bits uno por uno hasta que se hayan enviado los 8 bits completos.

¿Qué hace el puerto PORTB, 0?

El pin **PORTB, 0** está actuando como la **línea de datos** serial (normalmente llamada **SD**, *Serial Data*) en un protocolo de comunicación serial. A través de esta línea, el microcontrolador envía bits de información al display, donde:

- Un **0** enviado en **PORTB, 0** significa "apaga un segmento".
- Un **1** enviado en **PORTB, 0** significa "enciende un segmento".

Estos bits **0** y **1** se procesan dentro del display de 7 segmentos para encender o apagar las luces correspondientes a cada segmento del número que quieres mostrar.

¿Cómo se usan esos 0 y 1?

En el display de 7 segmentos, los bits enviados determinan qué segmentos del número se deben encender o apagar. Cada número del 0 al 9 en el display tiene un patrón específico de segmentos que deben estar encendidos.

Por ejemplo:

- **Número 0**: Se deben encender 6 segmentos (a, b, c, d, e, f).
- **Número 1**: Solo se deben encender los segmentos **b** y **c**.

Estos patrones se representan con bits. Para el **número 0**, el patrón de encendido podría verse así:

```
a b c d e f g (h es el punto decimal)
0 0 0 0 0 0 1 1  (el bit 0 apaga y el bit 1 enciende)
```

Si el bit es **0**, el segmento correspondiente se apaga, y si el bit es **1**, el segmento correspondiente se enciende. Estos bits son los que estás enviando a través de `PORTB, 0`.

¿A dónde van estos bits?

Los bits se van enviando, uno por uno, a través de la línea de datos **PORTB, 0** hacia el display de 7 segmentos, pero no basta con enviarlos. Se deben sincronizar con un **reloj** que indique cuándo el display debe leer y almacenar esos bits.

Este reloj lo controlas con el pin **PORTD, 7**, que actúa como la señal de **SCLK** (Serial Clock). Cada vez que se cambia el estado de **PORTD, 7** (de 0 a 1), se le

dice al display que lea el siguiente bit que está en la línea de datos (**PORTB, 0**). Así, bit a bit, el display recibe toda la información necesaria para saber qué segmentos encender.

¿Para qué sirven los bits que se envían?

Estos bits determinan qué números o símbolos se van a mostrar en el display. Por ejemplo:

- Si envías un patrón de bits que representa el número **0**, el display encenderá los segmentos correspondientes al **0**.
- Si envías el patrón para el **1**, encenderá solo los segmentos necesarios para mostrar el **1**.

Descripción de la rutina `dato_serie`:

1. `lsr r16`: Aquí se rota el registro **r16** a la derecha, y el bit menos significativo (el bit 0 de **r16**) se coloca en el bit de acarreo (**C**). Este bit será el que se envíe al display.
2. `brcs loop_dato2`: Verifica si el bit en **C** es 1. Si es 1, salta a **loop_dato2**, donde se ejecuta `sbi PORTB, 0` (esto significa que se envía un **1** al display).
3. Si el bit es **0**, no se salta y se ejecuta `cbi PORTB, 0`, lo que envía un **0** al display.
4. `sbi PORTD, 7`: Luego, se pone en alto el reloj serial (**SCLK**), lo que le indica al display que lea el bit que acabas de enviar (ya sea 0 o 1).
5. `cbi PORTD, 7`: Finalmente, el reloj se pone en bajo nuevamente, completando el ciclo para ese bit.

Este proceso se repite para todos los 8 bits de **r16**, enviando un byte completo al display de 7 segmentos.

Ejemplo:

Supongamos que quieres mostrar el número **0** en el display. El patrón de segmentos para el número **0** es **0b00000011** (se encienden los segmentos a, b, c, d, e, f, y se apaga el segmento g). Este valor de bits lo cargamos en **r16**, y bit a bit se envía al display:

1. Se rota el registro **r16**, enviando primero el bit menos significativo.
2. Si es un **0**, se ejecuta `cbi PORTB, 0` (se apaga el segmento correspondiente).

3. Si es un **1**, se ejecuta `sbi PORTB, 0` (se enciende el segmento correspondiente).
4. El reloj **SCLK** (controlado por **PORTD, 7**) sincroniza el envío de estos bits, asegurando que el display los lea correctamente.

¿Qué hace bien este código?

- **Serialización de datos**: El código convierte los valores binarios (almacenados en `r16` y `r17`) en una secuencia de bits que son enviados de manera serial al display de 7 segmentos. Los datos son enviados con un **protocolo de reloj y datos** (controlados por los pines `SCLK` y `SD`).
- **Control de actualización del display**: La rutina `sacanum` asegura que el display reciba los datos correctos para mostrar el número en el dígito correcto, usando un **latch** para fijar los datos en el momento apropiado.
- **Uso eficiente de registros**: El código utiliza los registros de manera eficiente para manejar tanto los bits de los segmentos como la selección del dígito activo.

9. Interrupciones de botones

```
pcint:
    in r26, SREG
    sbis PINC, 1
    ldi r29, 0b00000010
    sbis PINC, 2
    ldi r29, 0b00000100
    sbis PINC, 3
    call reset_contador
    rjmp pcint_out
pcint_out:
    out SREG, r26
    reti
```

- Esta interrupción responde al presionar los botones conectados al puerto C, verificando su estado y tomando acciones dependiendo de cuál botón ha sido presionado.
- `in r26, SREG`: Guarda el registro de estado en `r26`.

- `sbis PINC, 1`: Verifica si el botón conectado al pin PC1 está presionado (si el bit es 0).
- Si el botón es presionado, el código guarda en `r29` valores que indican qué botón fue presionado, y si es necesario, se llama a la rutina `reset_contador` para reiniciar los contadores.
- Los valores en `r29` (`0b00000010` y `0b00000100`) representan qué botón fue presionado.
- Esto permite que otras partes del código interpreten el estado de los botones y tomen acciones basadas en cuál fue presionado.
- El valor en `r29` actúa como un **marcador binario** para identificar el botón que activó la interrupción.
- `r29` **se compara con el bit 2** porque ahí es donde se guarda la información de que el botón en `PINC1` fue presionado.
- La instrucción `sbrs` se usa para verificar si ese bit está activado (si el botón fue presionado).
- Si el bit está activado, el programa continúa y el cronómetro avanza mostrando el número correspondiente en el display de 7 segmentos.
- **la configuración que hace que el sistema comience a contar cuando se presiona el botón más a la izquierda está en la rutina de interrupción** `pcint`. Cuando el botón en `PINC1` (puerto C, pin 1) se presiona, se cambia el valor en `r29` y el estado del contador se ajusta en consecuencia para que empiece a contar.

1. **Presión del botón en** `PINC1`:
   - Cuando presionas el botón en el pin `PINC1`, la rutina de interrupción de botones (`pcint`) se ejecuta.
   - En esta rutina, el registro `r29` se carga con el valor `0b00000010`, lo que indica que el botón en `PINC1` fue presionado.
2. **Interrupción del temporizador**:
   - Después de que pasa 1 segundo (configurado por el temporizador), se ejecuta la rutina de interrupción del temporizador (`_tmr0_int`).

- Dentro de esta rutina, si el contador de segundos (`contador_segundos`) llega a 0, se llama a la subrutina `_1hz`.

3. **Verificación del valor de `r29` en la rutina `_1hz`**:
   - Aquí, el código usa la instrucción `sbrs r29, 2`, que significa: **si el bit 2 de `r29` es 1, salta la siguiente instrucción**.
   - Esto verifica si el bit 2 de `r29` está en 1. Si **NO está en 1**, entonces el cronómetro continúa y se incrementa el valor de `estado_botones`, que representa el número que se mostrará en el display.
4. **Mostrar el número**:
   - A medida que se incrementa `estado_botones`, este valor se utiliza para actualizar el número que se mostrará en el display, gracias a la rutina `sacanum`, que controla la salida del número al display de 7 segmentos.

10. Interrupción del temporizador

```
_tmr0_int:
    in r17, SREG
    inc r23
    call act_display
    dec r25
    breq _1hz
    rjmp _tmr0_out
_tmr0_out:
    out SREG, r17
    reti
_1hz:
    sbi PINB, 2
    sbrs r29,2
    inc r19
    ldi r25, 250
    rjmp _tmr0_out
```

- Esta rutina se ejecuta cada vez que el temporizador alcanza el valor de comparación (`OCR0A`), actualiza el display y genera una señal de 1 Hz (aproximadamente).
- **Uso de `r17` en lugar de `r16`**:
- A lo largo del código, el registro `r16` parece estar más involucrado en otras operaciones (como el manejo del display), mientras que `r17` se utiliza temporalmente para guardar el contexto de `SREG` y luego restaurarlo al final de la interrupción.
- `r16` se usa frecuentemente para enviar datos al display en otras partes del código. Si guardaras el `SREG` en `r16`, lo sobreescribirías y perderías esos valores cuando `r16` se usara más adelante. Esto causaría errores.
- `r17` se reserva aquí como un registro temporal que no interfiere con las demás operaciones. Se usa exclusivamente para guardar el contexto al inicio de la interrupción y restaurarlo al final.

```
inc valor_display
```

Incrementar `valor_display` te permite controlar cuál de los dígitos del display se debe encender en cada momento.

¿Por qué es útil?

- Sin este incremento, sólo podrías encender un dígito del display a la vez.
- Incrementar `valor_display` permite mostrar varios números en el display, controlando cada dígito de manera secuencial.

TEMPORIZACION A 1 HZ:

1 Hz: Sí, 1 Hz corresponde a un segundo. En la rutina `_tmr0_int`, el código incrementa el contador `r23` y, cuando llega a un valor predeterminado, llama a `act_display` para actualizar la pantalla. El valor de `r25` controla cuántas veces tiene que decrementar antes de llegar a 1 Hz (en este caso, 250 ciclos para un segundo).

Relación con otras operaciones

Cada vez que ocurre la interrupción, puedes utilizarla para actualizar algo (en este caso, un contador o un display). En este código, el temporizador se utiliza para contar de manera más rápida (más de una vez por segundo) y luego se combina con otro contador (como `r25`), que cuenta cuántas veces ha ocurrido esta interrupción para generar un evento a 1 Hz (un segundo), acumulando estas interrupciones más rápidas.

Por ejemplo:

- `dec r25` y `breq _1hz`: Cada vez que ocurre la interrupción, `r25` se decrementa. Cuando llega a 0, se genera una señal de 1 segundo (1 Hz), lo que corresponde a 250 interrupciones (aproximadamente).

Explicación de `BRNE` y `BREQ`

1. `BRNE` **(Branch if Not Equal)**:
   - Esta instrucción realiza un **salto** a una dirección de memoria (etiqueta) **si el bit de la bandera de cero (Z) está en 0**, lo que indica que la operación anterior **no resultó en cero**.
   - Se utiliza típicamente para verificar si dos valores **no son iguales** o si el resultado de una operación **no es cero**.

   Sintaxis:

   ```
   brne etiqueta
   ```

   Esto significa: *salta a* `etiqueta` *si el resultado de la última operación no fue igual a cero (Z = 0)*.

2. `BREQ` **(Branch if Equal)**:
   - Esta instrucción realiza un **salto** si el **bit de la bandera de cero (Z) está en 1**, lo que indica que la operación anterior **dio un resultado igual a cero**.
   - Se usa típicamente para verificar si dos valores **son iguales** o si el resultado de una operación **es cero**.

   Sintaxis:

```
assembly
Copiar código
breq etiqueta
```

Esto significa: *salta a* `etiqueta` *si el resultado de la última operación fue igual a cero (Z = 1)*.

Uso y propósito

Ambas instrucciones se utilizan comúnmente después de operaciones de comparación (`CP`, `CPI`) o sustracción (`SUB`, `SUBI`) para tomar decisiones basadas en el resultado. Estas operaciones modifican el bit de bandera de cero (Z), que indica si los operandos comparados o el resultado de la operación es cero.

- `BRNE` **(Branch if Not Equal)** se usa para continuar el flujo del programa si los valores comparados **no son iguales** o si el resultado de una operación no fue cero.

  Ejemplo:

  ```
  assembly
  Copiar código
  cp r16, r17         ; Compara r16 con r17
  brne no_son_iguales ; Si no son iguales, salta a no_son_
  iguales
  ```

- `BREQ` **(Branch if Equal)** se usa para saltar si los valores comparados **son iguales** o si el resultado de una operación fue cero.

  Ejemplo:

  ```
  assembly
  Copiar código
  cp r16, r17         ; Compara r16 con r17
  breq son_iguales    ; Si son iguales, salta a son_iguale
  s
  ```

Ejemplo práctico

Supongamos que tienes un contador que debe realizar una acción cuando llega a cero:

```
assembly
Copiar código
dec r16             ; Decrementa r16
brne loop           ; Si r16 no es cero, repite el bucle
```

- **Aquí** `BRNE` es útil para mantener el bucle activo hasta que el contador (`r16`) llegue a cero.

O si estás comparando dos registros para ver si son iguales:

```
assembly
Copiar código
cp r16, r17         ; Compara r16 con r17
breq iguales        ; Si son iguales, salta a la etiqueta `
iguales`
```

- **Aquí** `BREQ` salta a `iguales` si los registros contienen el mismo valor, haciendo el flujo del programa más claro.

Conclusión

- `BRNE` se usa cuando deseas **tomar una acción si los valores no son iguales** o el resultado de una operación no fue cero.
- `BREQ` se usa cuando deseas **tomar una acción si los valores son iguales** o el resultado de una operación fue cero.

11. Actualización del display

```
act_display:
    cpi r23, 1
    breq pos_uno
```

```
    cpi r23, 2
    breq pos_dos
    cpi r23, 3
    breq pos_tres
    brne pos_cuatro
    ret
```

- Dependiendo del valor del contador `r23`, se actualiza la posición en el display que debe encenderse.

La rutina `act_display` determina qué dígito del display de 7 segmentos debe ser actualizado, dependiendo del valor de

```
r23
```

. Compara el valor de

```
r23
```

con valores fijos para cada posición del display:

- `breq pos_uno`, `breq pos_dos`, etc., seleccionan la posición del display a actualizar.

Luego llama a una rutina de traducción (`traductor`) que convierte los valores numéricos en los segmentos correspondientes para mostrarlos en el display.

Qué hace `act_display`?

1. **Determina cuál dígito del display actualizar**:
   La subrutina compara el valor de `valor_display`, que está siendo incrementado en la interrupción del Timer (`_tmr0_int`). Según el valor de `valor_display`, se activa un dígito diferente del display para mostrar el número correspondiente.

   ```
   asm
   Copiar código
   cpi valor_display, 1
   breq pos_uno      ; Si valor_display = 1, activa el prim
   er dígito
   cpi valor_display, 2
   ```

```
breq pos_dos      ; Si valor_display = 2, activa el segu
ndo dígito
cpi valor_display, 3
breq pos_tres     ; Si valor_display = 3, activa el terc
er dígito
brne pos_cuatro   ; Si no coincide con los anteriores, a
ctiva el cuarto dígito
ret
```

2. **Selecciona el dígito correspondiente**:
   Dependiendo del valor de `valor_display`, la subrutina llama a funciones específicas como `unidades`, `decenas`, `unidades_m` y `decenas_m`, que preparan los valores a ser mostrados en el display (como el número a mostrar). Luego, la subrutina `sacanum` se encarga de enviar el número al display utilizando comunicación serial.
   - `pos_uno`: Muestra las unidades del valor actual en el primer dígito.
   - `pos_dos`: Muestra las decenas en el segundo dígito.
   - `pos_tres`: Muestra las centenas en el tercer dígito.
   - `pos_cuatro`: Muestra los millares en el cuarto dígito.
3. **Control del multiplexado**:
   Al activar y desactivar los diferentes dígitos secuencialmente, la subrutina permite que los 4 dígitos del display se vean como si estuvieran encendidos simultáneamente, aunque en realidad se encienden uno por uno muy rápidamente. Este efecto se logra alternando entre los dígitos en las diferentes interrupciones del Timer.
4. En pocas palabras, `**act_display**` es responsable de controlar qué número aparece en qué posición del display, logrando que el display de 7 segmentos muestre múltiples dígitos de forma fluida mediante multiplexado.

Información necesaria para `sacanum`:

- `r16`: Contiene el **patrón de segmentos** que representa el número a mostrar (es decir, los LEDs a encender para formar el número en el display de 7 segmentos).

- `r17`: Contiene el **dígito** en el que se mostrará el número (es decir, cuál de los 4 dígitos del display debe estar activo).

Ambos registros se preparan **antes** de llamar a `sacanum`.

Análisis de `pos_uno`:

Cuando la subrutina `pos_uno` se llama, hace lo siguiente:

1. `ldi r17, 0b00010000`:
   - `r17` se carga con el valor **0b00010000**, que indica que el número se debe mostrar en el **dígito más a la derecha** del display.
2. `call unidades`:
   - Llama a `unidades`, que prepara el valor de `r16`. En `unidades`, se consulta el valor actual de las **unidades** (almacenado en `estado_botones`) y se convierte en un patrón de segmentos en **r16** usando la subrutina `traductor`.
   - Por ejemplo, si el valor de `estado_botones` es 5, `traductor` convierte este valor a `0b01001001` (que enciende los segmentos adecuados para mostrar el número 5 en el display).
3. `call sacanum`:
   - Después de preparar `r16` y `r17`, se llama a `sacanum` para **enviar** estos valores al display.

Análisis de `sacanum`:

```
assembly
Copiar código
sacanum:
    call dato_serie          ; Envía el patrón de segmentos
(r16) serialmente al display
    mov  r16, r17            ; Mueve el valor de r17 (posic
ión del dígito) a r16
    call dato_serie          ; Envía la posición del dígito
(r16) serialmente al display
    sbi  PORTD, 4            ; Latch: activa el dígito del
display (PD4 = 1)
    cbi  PORTD, 4            ; Latch: desactiva el dígito d
```

```
el display (PD4 = 0)
    ret
```

- `call dato_serie` (primera vez): Se envía el contenido de `r16` (el patrón de segmentos, que representa el número a mostrar) bit por bit al display.
- `mov r16, r17`: Después de enviar el número, se copia el valor de `r17` (la posición del dígito) a `r16`. Esto es necesario porque `dato_serie` siempre trabaja con `r16`, y ahora necesitamos enviar el valor que estaba en `r17` (la posición del dígito).
- `call dato_serie` (segunda vez): Se envía el valor de `r16` (que ahora contiene la posición del dígito) serialmente al display.
- `sbi PORTD, 4` y `cbi PORTD, 4`: Estas instrucciones activan y desactivan la señal de **latch** (PD4), que indica al display que debe actualizar el contenido del dígito que se seleccionó. Esto asegura que el valor enviado se muestre en el dígito correcto.

Información necesaria para `sacanum`:

- `r16`: Contiene el **patrón de segmentos** que representa el número a mostrar (es decir, los LEDs a encender para formar el número en el display de 7 segmentos).
- `r17`: Contiene el **dígito** en el que se mostrará el número (es decir, cuál de los 4 dígitos del display debe estar activo).

Ambos registros se preparan **antes** de llamar a `sacanum`.

Análisis de `pos_uno`:

Cuando la subrutina `pos_uno` se llama, hace lo siguiente:

1. `ldi r17, 0b00010000`:
    - `r17` se carga con el valor **0b00010000**, que indica que el número se debe mostrar en el **dígito más a la derecha** del display.
2. `call unidades`:
    - Llama a `unidades`, que prepara el valor de `r16`. En `unidades`, se consulta el valor actual de las **unidades** (almacenado en `estado_botones`) y se

convierte en un patrón de segmentos en **r16** usando la subrutina `traductor`.

- Por ejemplo, si el valor de `estado_botones` es 5, `traductor` convierte este valor a `0b01001001` (que enciende los segmentos adecuados para mostrar el número 5 en el display).

3. `call sacanum`:
   - Después de preparar `r16` y `r17`, se llama a `sacanum` para **enviar** estos valores al display.

Análisis de `sacanum`:

```
sacanum:
    call dato_serie          ; Envía el patrón de segmentos
(r16) serialmente al display
    mov  r16, r17            ; Mueve el valor de r17 (posic
ión del dígito) a r16
    call dato_serie          ; Envía la posición del dígito
(r16) serialmente al display
    sbi  PORTD, 4            ; Latch: activa el dígito del
display (PD4 = 1)
    cbi  PORTD, 4            ; Latch: desactiva el dígito d
el display (PD4 = 0)
    ret
```

Resumen General del Proceso en `sacanum`:

1. `r16` contiene el **patrón de segmentos** (el número a mostrar), por ejemplo, el valor que representa el número **0**.
2. `r17` contiene la **posición** (qué dígito del display se debe actualizar), por ejemplo, el dígito más a la derecha.
3. La subrutina `sacanum` envía estos dos valores, **bit por bit**, al **display** a través de la subrutina `dato_serie`. Primero, envía el patrón de segmentos (el número) y luego la posición (el dígito en el display).

Paso a Paso del Flujo de `sacanum`:

1. Enviar el Patrón de Segmentos (`r16`) a través de `dato_serie` :

Cuando `sacanum` es llamada, lo primero que hace es enviar el **contenido de** `r16`, que contiene el patrón de segmentos para el número que quieres mostrar, bit por bit.

```
call dato_serie    ; Envía el número (contenido de r16) ser
ialmente al display
```

- `dato_serie` lee los bits de `r16` uno por uno, comenzando desde el bit menos significativo(porque este es un patrón común en la **comunicación serial** de muchos dispositivos electrónicos), y los envía a través del pin de datos (probablemente **PORTB, 0**).
- Al mismo tiempo, `dato_serie` utiliza el pin de reloj **SCLK** (controlado por **PORTD, 7**) para sincronizar la transmisión de los bits.
- Cada ciclo de reloj (cada vez que `sbi PORTD, 7` y `cbi PORTD, 7` alternan entre 0 y 1), el display recibe un bit del valor de `r16`.

En este punto, el display está recibiendo el **número** (por ejemplo, el patrón **0b00000011** para el número 0), pero aún no sabe en qué **dígito** del display debe mostrarse.

2. Enviar la Posición del Dígito (`r17`) a través de `dato_serie` :

Una vez que se ha enviado el número, `sacanum` copia el contenido de `r17` (la posición del dígito) a `r16`, para enviarlo también bit por bit al display.

```
mov r16, r17       ; Mueve el valor de r17 (posición del dí
gito) a r16
call dato_serie    ; Envía la posición (contenido de r17) s
erialmente al display
```

- `dato_serie` se llama nuevamente, pero ahora el valor en `r16` (copiado de `r17`) es la **posición del dígito** donde se debe mostrar el número.

- El proceso es el mismo que antes: `dato_serie` envía este valor **bit por bit** usando el mismo pin de datos y el reloj serial.

3. Activar el Latch para Actualizar el Display:

Después de enviar tanto el **número** como la **posición del dígito**, `sacanum` activa el **latch** del display mediante **PORTD, 4**. Este latch le indica al display que debe **actualizar el dígito seleccionado** con el número que acaba de recibir.

```
sbi PORTD, 4        ; Activa el latch (PD4 = 1) para actuali
zar el dígito
cbi PORTD, 4        ; Desactiva el latch (PD4 = 0)
```

Detalles de `dato_serie` (Transmisión Bit a Bit):

`dato_serie` es la subrutina que gestiona el envío de datos seriales. Su tarea es **enviar los 8 bits** del registro `r16` al display, uno por uno. Cada vez que se llama a `dato_serie`, esta rutina:

1. Rota el valor de `r16` a la derecha (`lsr r16`), de modo que el bit menos significativo se coloca en el bit de acarreo (**C**).
2. Dependiendo de si el bit de acarreo es 0 o 1, escribe ese valor en el pin de datos (**PORTB, 0**):
   - Si el bit es **0**, se ejecuta `cbi PORTB, 0`.
   - Si el bit es **1**, se ejecuta `sbi PORTB, 0`.
3. Alterna el pin de reloj (**SCLK** en **PORTD, 7**), para que el display sepa cuándo leer el siguiente bit.

Este proceso se repite 8 veces para enviar todos los bits de `r16`.

Análisis de `sacanum`:

```
sacanum:
    call dato_serie          ; Envía el patrón de segmentos
(r16) serialmente al display
    mov  r16, r17            ; Mueve el valor de r17 (posic
```

```
ión del dígito) a r16
    call dato_serie          ; Envía la posición del dígito
(r16) serialmente al display
    sbi  PORTD, 4            ; Latch: activa el dígito del
display (PD4 = 1)
    cbi  PORTD, 4            ; Latch: desactiva el dígito d
el display (PD4 = 0)
    ret
```

- `call dato_serie` (primera vez): Se envía el contenido de `r16` (el patrón de segmentos, que representa el número a mostrar) bit por bit al display.
- `mov r16, r17`: Después de enviar el número, se copia el valor de `r17` (la posición del dígito) a `r16`. Esto es necesario porque `dato_serie` siempre trabaja con `r16`, y ahora necesitamos enviar el valor que estaba en `r17` (la posición del dígito).
- `call dato_serie` (segunda vez): Se envía el valor de `r16` (que ahora contiene la posición del dígito) serialmente al display.
- `sbi PORTD, 4` y `cbi PORTD, 4`: Estas instrucciones activan y desactivan la señal de **latch** (PD4), que indica al display que debe actualizar el contenido del dígito que se seleccionó. Esto asegura que el valor enviado se muestre en el dígito correcto.

¿Cómo sabe `sacanum` qué número y posición mostrar?

1. **Número a mostrar**:
    - El número se almacena en `r16` antes de llamar a `sacanum`. Este registro se llena en la subrutina `unidades` (o en las subrutinas de decenas, centenas, etc.) usando el valor de `estado_botones` o de otros contadores.
    - El valor en `r16` es el patrón de bits que enciende los segmentos del display para representar el número deseado (por ejemplo, `0b01001001` para el número 5).
2. **Dígito del display**:
    - La posición del dígito donde se mostrará el número está en `r17`. Este valor lo establece `pos_uno` (o las otras subrutinas como `pos_dos`,

`pos_tres`, etc.) para indicar qué dígito (de los 4 disponibles) debe actualizarse.

- `r17` tiene valores como `0b00010000` (para el dígito más a la derecha), `0b00100000` (para el segundo dígito de derecha a izquierda), y así sucesivamente.

Resumen del flujo:

1. **Antes de llamar a `sacanum`:**
   - `pos_uno` (u otras subrutinas como `pos_dos`) carga el valor de `r17` con la **posición del dígito** donde se mostrará el número.
   - La subrutina `unidades` carga `r16` con el **patrón de segmentos** correspondiente al número que se quiere mostrar.
2. **Cuando se llama a `sacanum`:**
   - `sacanum` envía primero el valor de `r16` (el patrón de segmentos) al display, para que los segmentos correctos se enciendan y representen el número.
   - Luego, envía el valor de `r17` (la posición del dígito) para activar el dígito correcto donde se debe mostrar el número.
   - Finalmente, usa la señal de **latch** (controlada por **PORTD, 4**) para que el display actualice el dígito seleccionado.

Resumen:

- `r16` guarda el **número a mostrar** en forma de un patrón de segmentos (los LEDs a encender/apagar en el display).
- `r17` guarda la **posición del dígito** en el que se va a mostrar el número.
- `sacanum` utiliza estos registros para enviar la información serialmente al display, asegurando que el número correcto aparezca en el dígito correcto.

```
_1hz:
sbi PINB, 2
sbrs r29,2
inc estado_botones
ldi contador_segundos, 250
rjmp _tmr0_out
```

- **Si** `r29` **bit 2 = 1, se salta** la instrucción `inc estado_botones` y no pasa nada.
- **Si** `r29` **bit 2 = 0, se ejecuta** la instrucción `inc estado_botones`, incrementando el valor de `estado_botones`.

Esto te permite controlar condicionalmente si `estado_botones` se incrementa o no, dependiendo del valor del bit 2 de `r29`.

- SIRVE PARA QUE EL TIMER EMPIECE APAGADO.

Análisis del código de la interrupción:

1. `sbis PINC, 2`: Este comando verifica si el botón en **PINC2** está presionado. Si **no** está presionado, se salta la siguiente línea.
2. `ldi r29, 0b00000100`: Si el botón en **PINC2** está presionado, se carga el valor `0b00000100` en el registro `r29`. Este valor indica que **el bit 2 de** `r29` está activado.

¿Cómo detiene el cronómetro cuando se presiona `PINC2`?

El valor almacenado en `r29` (después de presionar el botón en `PINC2`) afecta el comportamiento del cronómetro en la **rutina del temporizador**, específicamente en la subrutina `_1hz`, que se ejecuta cada vez que pasa un segundo. Aquí está el código relevante:

```
_1hz:
    sbi PINB, 2                ; Activa un LED o marca que
pasó 1 segundo
    sbrs r29, 2                ; Si el bit 2 de r29 es 0, s
alta la siguiente instrucción
    inc estado_botones         ; Si el bit 2 de r29 no es
0, incrementa estado_botones (el cronómetro sigue)
    ldi contador_segundos, 250  ; Reinicia el contador de s
egundos
    rjmp _tmr0_out
```

- `sbrs r29, 2`: Esta instrucción significa "**salta si el bit 2 de** `r29` **está activado**".

- Si presionaste el botón en **PINC2**, el bit 2 de `r29` está en 1 (`0b00000100`), y esto provoca que el código **salte la instrucción** `inc estado_botones`.
- Al saltarse `inc estado_botones`, el valor de `estado_botones` (que representa el tiempo contado) **no se incrementa**, lo que significa que **el cronómetro se detiene** porque no se avanza más en el conteo.

El temporizador sigue generando interrupciones independientemente, pero lo que **detiene el cronómetro** es la **evaluación del estado de los botones (especialmente el valor de `r29`)**, y eso se hace típicamente dentro de la rutina `_1hz`, que se ejecuta cada segundo.

Por lo tanto, cuando presionas **PCIN2**, el cronómetro **no se detiene inmediatamente**, pero el estado de `r29` se actualiza. Luego, en la próxima ejecución de la rutina `_1hz` (que ocurre cuando el temporizador ha contado 1 segundo completo), el programa evalúa `r29` y puede decidir detener el cronómetro basándose en el estado del botón.

¿Por qué parece detenerse tan rápido?

Aunque el temporizador genera interrupciones cada 4 ms, la evaluación del estado del cronómetro se hace cada segundo dentro de `_1hz`. Si presionas el botón poco antes de que se ejecute la rutina `_1hz`, **parece que el cronómetro se detiene de inmediato** porque la condición del botón es evaluada poco después de presionarlo.

Detalles técnicos:

- `sbrs`: Significa "Skip if Bit in Register is Set", es decir, "Saltar si el bit en el registro está activado". En este caso, se verifica el bit 2 de `r29`.
  - Si el bit 2 de `r29` está en 1 (porque presionaste `PINC2`), la instrucción `inc estado_botones` se salta, y el cronómetro no avanza.
- **Efecto del botón** `PINC2`: Cuando presionas el botón, no se incrementa `estado_botones`, por lo que el número que se muestra en el display se **mantiene igual**, deteniendo el cronómetro.

```
unidades:
mov r27, estado_botones  ; Copia el valor de las unidades (es
cpi r27, 10              ; Compara si el valor de las unidade
brne traductor           ; Si no es 10, salta a 'traductor' p
```

```
call desborde_unidades   ; Si es 10, llama a 'desborde_unidad
ret
```

- `mov r27, estado_botones`: Se mueve el valor de `estado_botones` a `r27`. Este registro contiene el valor actual de las unidades (0-9).
- `cpi r27, 10`: Se compara si el valor de **r27** (es decir, las unidades) es igual a 10. Esto se hace porque el máximo valor de las unidades es 9; cuando alcanza 10, se produce un **desborde**.
- `brne traductor`: Si las unidades no han alcanzado 10 (no hay desborde), se salta a la subrutina `traductor`, que convertirá el valor de las unidades en un patrón de segmentos para el display.
- `call desborde_unidades`: Si el valor de las unidades es igual a 10, se llama a la subrutina `desborde_unidades` para gestionar el desborde, es decir, para incrementar las **decenas** y reiniciar las unidades a 0.

```
desborde_unidades:
inc contador_decenas    ; Incrementa el valor de las decenas
ldi estado_botones, 0   ; Reinicia las unidades a 0
mov r27, estado_botones  ; Copia el nuevo valor de las unidad
call traductor           ; Llama a la rutina 'traductor' para
ret
```

- `inc contador_decenas`: Aquí se incrementa el registro `contador_decenas`, que guarda el valor de las decenas. Esto ocurre cuando las unidades llegan a 10, lo que significa que ahora hay que sumar 1 a las decenas.
- `ldi estado_botones, 0`: El valor de las unidades se reinicia a **0** (ya que el valor 10 se representa como "1 decena y 0 unidades").
- `mov r27, estado_botones`: Se mueve el valor de `estado_botones` (que ahora es 0) al registro `r27` para que la subrutina `traductor` lo procese y lo convierta en el patrón de segmentos para el número 0 en el display.
- `call traductor`: Finalmente, se llama a la subrutina `traductor`, que traduce el valor numérico en `r27` (en este caso, 0) a un patrón de segmentos para el display.

¿Qué sucede cuando hay un desbordamiento en las unidades?

Cuando el valor en las **unidades** llega a 10, se produce un **desbordamiento**. Aquí es cuando las decenas se incrementan y las unidades vuelven a 0. A continuación te explico cómo ocurre:

1. **Unidades llegan a 10**:
   - La rutina `unidades` detecta que `estado_botones` es igual a 10. Esto dispara la llamada a la rutina `desborde_unidades`.

```
unidades:
    mov r27, estado_botones
    cpi r27, 10
    brne traductor           ; Si no es 10, traduce norm
almente.
    call desborde_unidades   ; Si es 10, ocurre desborda
miento.
    ret
```

2. **Desborde de Unidades**:
   - En la rutina `desborde_unidades`, ocurre lo siguiente:
     - **Incremento de decenas**: Se incrementa `contador_decenas`, que contiene el valor de las decenas.
     - **Reinicio de las unidades**: `estado_botones` (que contiene las unidades) se reinicia a 0.
     - Se llama a `traductor` para obtener la representación correcta de las unidades (ahora 0) en el display.

```
assembly
Copiar código
desborde_unidades:
    inc contador_decenas      ; Incrementa el valor de l
as decenas
    ldi estado_botones, 0     ; Reinicia las unidades a
0
    mov r27, estado_botones
    call traductor            ; Traduce el valor de las
```

```
unidades (0)
    ret
```

3. **Próximos pasos tras el desbordamiento**:
   - **Unidades**: En la siguiente interrupción de `Timer0`, cuando `valor_display` sea 1 (unidades), el display mostrará **0** porque las unidades se reiniciaron a 0.
   - **Decenas**: En la siguiente vuelta, cuando `valor_display` **sea 2 (decenas)**, se mostrará el valor actualizado de las decenas. Esto se maneja en la rutina `decenas`.

     ```
     assembly
     Copiar código
     pos_dos:
         ldi r17, 0b00100000     ; Selecciona el segundo d
     ígito (decenas)
         call decenas            ; Muestra el valor de las
     decenas
         call sacanum            ; Envía el valor al displ
     ay
         ret
     ```

   - En la rutina `decenas`, el valor de `contador_decenas` (que se incrementó en el desbordamiento) se traduce al formato correcto para el display de 7 segmentos:

     ```
     assembly
     Copiar código
     decenas:
         mov r27, contador_decenas   ; Carga el valor de l
     as decenas en r27
         cpi r27, 6                  ; Verifica si el valo
     r es 6 (para desbordar)
         brne traductor              ; Si no es 6, traduce
     el valor de las decenas
     ```

```
    call desborde_decenas       ; Si es 6, se maneja
el desbordamiento de decenas
    ret
```

- Luego, la rutina `traductor` convertirá ese valor de `contador_decenas` a la representación de los segmentos del display (0b10011111 para el número 1, por ejemplo, si las decenas son 1).

Recapitulación después del desbordamiento de unidades:

1. **Unidades** vuelven a 0 (por el desbordamiento).
2. **Decenas** se incrementan y muestran el valor actualizado en el siguiente ciclo de actualización del display (cuando `valor_display = 2`).
3. Los demás dígitos (centenas y millares) siguen siendo 0 hasta que ocurra un desbordamiento en las decenas o centenas.

12. Traducción de números a segmentos

Finalmente, la rutina `traductor` convierte los valores numéricos en los patrones correspondientes de bits para los segmentos del display:

```
traductor:
    cpi r27,0
    breq cero
    cpi r27,1
    breq uno
    ; ...
cero:
    ldi r16,0b00000011
    ret
uno:
    ldi r16,0b10011111
    ret
```

- Los patrones de bits determinan qué segmentos del display de 7 segmentos deben encenderse para mostrar los números del 0 al 9.

- La subrutina `traductor` convierte el valor en `r27` en el patrón de segmentos adecuado para el display de 7 segmentos. Esta parte de tu código ya está definida para los números del 0 al 9.

Por ejemplo, para el número 0, el patrón de segmentos es `0b00000011`, que enciende los segmentos a, b, c, d, e y f, dejando apagado el segmento g. Para el número 1, se encienden solo los segmentos b y c, con el patrón `0b10011111`.

Desglosemos esta situación:

1. **Centenas (`valor_display = 3`) y millares (`valor_display = 4`) no se actualizan** inmediatamente porque el programa aún no ha llegado a un punto en el que estas posiciones necesiten incrementarse.

   - **Centenas** (`contador_centenas`) solo se incrementan cuando hay un **desbordamiento en las decenas**.
   - **Millares** (`contador_millares`) solo se incrementan cuando hay un **desbordamiento en las centenas**.

   Es decir, al comienzo del programa, **centenas** y **millares** estarán en **0** y no cambiarán hasta que las unidades y decenas acumulen suficientes desbordamientos para afectar estos dígitos.

2. **El conteo empieza desde las unidades**:

   - Primero, el conteo ocurre en las **unidades** (`estado_botones`). Cuando `estado_botones` llega a **10**, se produce un **desbordamiento de unidades**, lo que **incrementa las decenas**.
   - Las **decenas** (`contador_decenas`) se incrementan cada vez que las unidades alcanzan 10. Solo después de que las **decenas** lleguen a 10, se producirá un **desbordamiento de decenas** que incrementará las **centenas**.
   - De manera similar, las **centenas** (`contador_centenas`) solo se incrementan cuando las decenas han completado un ciclo (llegan a 10), lo que produce un **desbordamiento de decenas**.
   - Finalmente, los **millares** (`contador_millares`) solo se incrementan cuando las centenas se desbordan.

Resumen del proceso de actualización:

- **Al principio**, los dígitos **centenas** y **millares** no cambian porque el conteo **aún no ha alcanzado un valor que afecte esos dígitos**.
- **Unidades** y **decenas** cambian primero. Solo cuando las **decenas** llegan a 10, las **centenas** se incrementan, y lo mismo ocurre para los **millares**.

Situación actual de tu programa:

1. **Al principio**:
   - Las **centenas** (`valor_display = 3`) y **millares** (`valor_display = 4`) no cambian, y no se muestran aún porque no ha habido suficiente conteo en unidades y decenas para afectarlas.
   - Cuando el programa llega a `valor_display = 3` o `valor_display = 4`, **el valor mostrado será 0**, ya que no ha habido ningún cambio en esos dígitos.
2. **Con el tiempo**:
   - Cuando las **unidades** y **decenas** hayan contado lo suficiente como para causar un desbordamiento en las **centenas**, entonces el valor de las **centenas** cambiará y se mostrará un valor diferente a 0.
   - Lo mismo ocurrirá con los **millares**, pero solo cuando las **centenas** también se desborden.