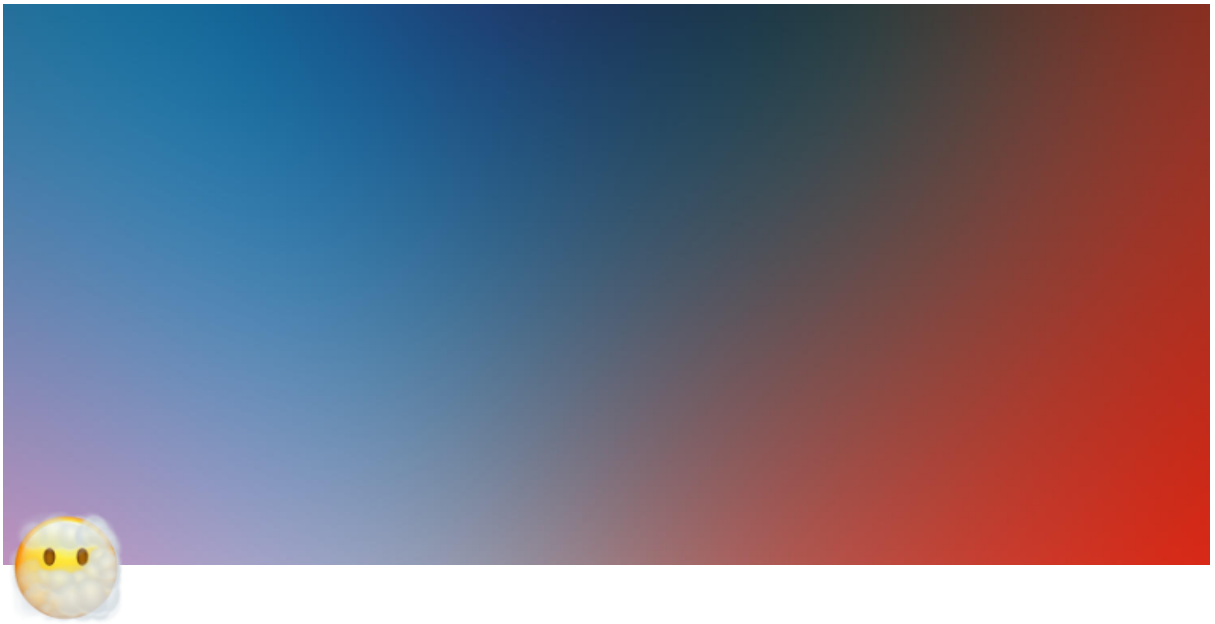

LABORATORIO 5 :FINAL

CODIGO DE LA MAESTRA:

```
; Laboratorio 5 , por ahora contiene las siguientes rutinas
;
; aleatorios - genera el vector de 512 números pseudoaleatori
;              XORSHIFT de 32 bits (https://en.wikipedia.org/
```

```
;
; Chksum_512 - Calcula el checksum de los 512 bytes en buffer
;              guarda el resultado en checksum_high:checksum_
;
```

```
; TX_512      - Transmite por el USART, los 1024 bytes de buf
;
; RX_512      - Recibe por el usart, los 1024 bytes.
;               que transmite la otra placa y lo coloca en buf
;
```

```
; _pcint1     - Rutina de atención a la interrupción de los b
```

```
;               si algún botón está apretado pone el bit0 de
```

```
;
; _tmr0_int   - Rutina de atención a la interupción del timer
;               segundo. Esta rutina saca por el disply check
```

```
;
;
; Registros reservados (uso global):
; checksum_high:checksum_low - Cheksum de 16 bits, es necesar
; r25             - Contiene el dígito en el display que esto
; r26             - Bit0: indica si se apretó un botón.
;
; Otros:
; r19:r18:r17:r16 - semilla de los números pseudoaleatorios .
;                   solo los usa aleatorios cuando está gener
```

```
; Definiciones de registros para facilitar su identificación
.def checksum_low = r4       ; r4 será llamado checksum_low
.def checksum_high = r5      ; r5 será llamado checksum_high
.def temp_reg = r1           ; r1 será llamado temp_reg
.def datoTransmitido = r0    ; Definición ya existente para r
```

```
; comienzo del código
.ORG 0x0000
jmp start         ; dirección de comienzo (vector de reset)
```

```
.ORG 0x0008
jmp _pcint1       ; salto a la rutina de atención a pcint1, i
.ORG 0x001C
jmp _tmr0_int     ; salto atención a rutina de comparación A
; ------------------------------------------------------------
```

```
; memoria RAM
.DSEG
buffer_msg: .byte 512         ; reservo 512 bytes para el vec
bmsg_end:   .byte 1           ; solo para marcar el final del
```

```
; comienzo del programa principal
.CSEG
start:
```

```
call system_init
ldi r26, 0x00             ; bandera teclado
mov checksum_high, r26    ; checksum a 0
mov checksum_low, r26
sei                       ; habilito interrupciones para disp
```

```
;jmp modo_transmisor
jmp modo_receptor
```

```
;modo_transmisor:
;   ldi r16, 0xA3             ; semilla de los números seudo-
;  ldi r17, 0x82
; ldi r18, 0xF0
;ldi r19, 0x05
;modo_transmisor_2:
;   rcall aleatorios          ; Genero los números aleatorios
```

```
;  rcall Chksum_512          ; Genero Checksum
```

```
; ldi r26, 0
;wait_4TX:                    ; espero que alguien presione
;   sbrs r26, 0               ; Nota: la interrupcion del bot
```

```
;  rjmp wait_4TX
```

```
; cli                       ; deshabilito interrupciones para
;rcall TX_512
;sei                       ; habilito interrupciones para dis
```

```
;rjmp modo_transmisor_2    ; empiezo todo de nuevo
```

```
;--------------------------------------
modo_receptor:
```

```
ldi     r26,    0
```

```
wait_4RX:                           ;acá me pongo a esperar q
sbrs    r26,    0               ;Nota: la interrupcion del bo
rjmp    wait_4RX
```

```
;ahora recibo 512 bytes y los dejo en buffer_msg
lds     r16,    UDR0            ;me aseguro que el buffer est
lds     r16,    UDR0
lds     r16,    UDR0
cli                             ;deshabilito interrupciones p
rcall   RX_512                      ;recibo 512 bytes por pol
sei                                 ;habilito interrupciones
rcall   Chksum_512                  ;calculo el nuevo Cheksum
rjmp    modo_receptor
```

```
;-------------------------------------------------------------
; Chksum - calcula el Checksum del vector buffer_msg (512 val
;-------------------------------------------------------------
Chksum_512:
ldi YL, low(buffer_msg)        ; Apunto Y al primer byte del
ldi YH, high(buffer_msg)
```

```
clr checksum_high              ; Inicializo el checksum
clr checksum_low
```

```
chksum_loop:
ld datoTransmitido, Y+         ; Traigo 1 byte a sumar
add checksum_low, datoTransmitido   ; Sumo el byte al checksu
adc checksum_high, temp_reg    ; Propago el acarreo a la part
cpi YL, low(bmsg_end)          ; Compruebo si llegué al final
brne chksum_loop
cpi YH, high(bmsg_end)
brne chksum_loop
ret
```

```
;-------------------------------------------------------------
; TX - rutina de transmisión serial USART. Transmite los 512
;-------------------------------------------------------------
;TX_512:
;   ldi ZL, low(buffer_msg)        ; Apunto Z al primer byte
;  ldi ZH, high(buffer_msg)
; ldi r16, (1 << TXEN0)          ; Habilito el transmisor USA
;sts UCSR0B, r16
```

```
;TX_loop1:
;   ld datoTransmitido, Z+         ; Traigo el Byte a transmi
;  sts UDR0, datoTransmitido      ; Pongo a transmitir (UDR0)
```

```
;TX_loop2:
;   lds r16, UCSR0A                ; Espero a que termine la
;  sbrs r16, UDRE0
; rjmp TX_loop2
```

```
;cpi ZL, low(bmsg_end)          ; Chequeo si llegué al fina
l del buffer
;brne TX_loop1
;cpi ZH, high(bmsg_end)
```

```
;brne TX_loop1

;ret
```

```
;chequeo si llegué al final del buffer
```

```
;ret
```

```
;-------------------------------------------------------------
; RX - rutina de recepción usart. Recibe 1024 bytes y los dej
;-------------------------------------------------------------
RX_512:
ldi ZL, low(buffer_msg)        ; Apunto Z al primer byte del
ldi ZH, high(buffer_msg)
ldi r16, (1 << RXEN0)          ; Configuro el USART como rece
sts UCSR0B, r16
```

```
RX_Wait:
lds     r16, UCSR0A               ; Ahora polling para esper
sbrs    r16, RXC0
rjmp    RX_Wait
```

```
lds     datoTransmitido, UDR0                   ; Llego aqu
í solo si recibí algo
st      Z+, datoTransmitido                     ; Guardo lo
que recibí

cpi     ZL, low(bmsg_end)          ; Chequeo si llegué al f
inal del buffer
brne    RX_Wait
cpi     ZH, high(bmsg_end)
brne    RX_Wait

ret
```

```
//-------------------------------------------
system_init:
;configuro los puertos:
;   PB2 PB3 PB4 PB5 - son los LEDs del shield
;   PB0 es SD (serial data) para el display 7seg
;   PD7 es SCLK, el reloj de los shift registers del display
;   PD4 transfiere los datos que ya ingresaron en serie, a la
```

```
ldi     r16,    0b00111101
out     DDRB,   r16         ;4 LEDs del shield son salidas
out     PORTB,  r16         ;apago los LEDs
ldi     r16,    0b00000000
out     DDRC,   r16         ;3 botones del shield son entra
das
ldi     r16,    0b10010001
out     DDRD,   r16         ;configuro PD.0, PD.4 y PD.7 co
mo salidas
cbi     PORTD,  7           ;PD.7 a 0, es el reloj serial d
el  Display, inicializo a 0
cbi     PORTD,  4           ;PD.4 a 0, es el reloj del latc
h del Display, inicializo a 0
```

```
;-----------------------------------------------------------
;Configuro interrupcion por cambio en PC.1, PC.2, PC.3.
ldi     r16,    0b00000010
sts     PCICR,  r16         ;habilito PCI1 que es (PCI8:PCI14
ldi     r16,    0b00001110
sts     PCMSK1, r16         ;habilito detectar cambios en Por
;-----------------------------------------------------------
;Configuro el TMR0 y su interrupcion.
ldi     r16,    0b00000010
out     TCCR0A, r16         ;configuro para que cuente hasta
ldi     r16,    0b00000010
out     TCCR0B, r16         ;prescaler = 256
ldi     r16,    249
out     OCR0A,  r16         ;comparo con 249
```

```
ldi     r16,    0b00000010
sts     TIMSK0, r16         ;habilito la interrupción (falta
;-------------------------------------------------------------
;Inicializo USART para transmitir
ldi     r16,    0b00010000
sts     UCSR0B, r16
ldi     r16,    0b00000110
sts     UCSR0C, r16
ldi     r16,    0x00            //9600 baudios (baudio = bit/
sts     UBRR0H, r16
ldi     r16,    0x67
sts     UBRR0L, r16
```

```
;-------------------------------------------------------------
;Inicializo algunos registros que voy a usar como variables.
ldi     r25,    0x10        ;inicializo r25 para el display r
;-------------------------------------------------------------
;Fin de la inicialización
ret
```

```
;-------------------------------------------------------------
;                   *           RUTINAS         *
;-------------------------------------------------------------
```

```
;-------------------------------------------------------------
; ALEATORIOS
;-------------------------------------------------------------
;rutina que genera 512 bytes pseudoaleatorios en buffer_msg
```

```
aleatorios:
ldi     r28,    low(buffer_msg)     ;apunto Y al primer byte
ldi     r29,    high(buffer_msg)
ale_loop:
; genero un número de 32bits nuevo usando XORSHIFT de 32 bits
ldi     r20,    13
call    ale_loop_l
```

```
ldi     r20,    17
call    ale_loop_r
ldi     r20,    5
call    ale_loop_l
;------ acá ya tengo el numero pseudo-aleatorio de 32bits, vo
```

```
st      Y+,     r16                 ;el número aleatorio lo
guardo a partir de adonde apunta el registro Y, voy recorri
endo hasta 512
st      Y+,     r17
st      Y+,     r18
st      Y+,     r19
cpi     YL,     low(bmsg_end)
brne    ale_loop
cpi     YH,     high(bmsg_end)
brne    ale_loop
ret
```

```
ale_loop_l:
mov     datoTransmitido,        r16
mov     r1,     r17
mov     r2,     r18
mov     r3,     r19
ale_rota_l:
clc
rol     datoTransmitido
rol     r1
rol     r2
rol     r3
dec     r20
brne    ale_rota_l
rjmp    ale_rota_out
```

```
ale_loop_r:
mov     datoTransmitido,        r16
mov     r1,     r17
```

```
mov     r2,     r18
mov     r3,     r19
ale_rota_r:
clc
ror     r3
ror     r2
ror     r1
ror     datoTransmitido
dec     r20
brne    ale_rota_r
```

```
ale_rota_out:
eor     r16,        datoTransmitido
eor     r17,        r1
eor     r18,        r2
eor     r19,        r3
ret
```

```
;-----------------------------------------------------------
;   SACANUM
;-----------------------------------------------------------
; rutina que saca un número por el display
```

```
sacanum:
push r16                ; guardo una copia de r16
ldi zh, high(segmap<<1) ; Initialize Z-pointer
ldi zl, low(segmap<<1)
andi r16, 0x0F
add zl, r16
clr r16
adc zh, r16
lpm r16, Z              ; traigo de la memoria de Programa el
call sacabyte
pop r16
call sacabyte
sbi PORTD, 4            ; PD.4 a 1, es el reloj del latch
```

```
cbi PORTD, 4            ; PD.4 a 0, es el reloj del latch
ret
```

```
;-----------------------------------------------------------
;   SACABYTE
;-----------------------------------------------------------
; saca un byte por el 7seg
sacabyte:
ldi r17, 0x08
loop_byte1:
cbi PORTD, 7            ; SCLK = 0
lsr r16
brcs loop_byte2         ; salta si C=1
cbi PORTB, 0            ; SD = 0
rjmp loop_byte3
loop_byte2:
sbi PORTB, 0            ; SD = 1
loop_byte3:
sbi PORTD, 7            ; SCLK = 1
dec r17
brne loop_byte1
ret
```

segmap:

```
.db 0b00000011, 0b10011111, 0b00100101, 0b00001101 ;"0" "1" "
.db 0b10011001, 0b01001001, 0b01000001, 0b00011111 ;"4" "5" "
.db 0b00000001, 0b00001001, 0b00010001, 0b11000001 ;"8" "9" "
.db 0b01100011, 0b10000101, 0b01100001, 0b01110001 ;"C" "d" "
```

```
; Rutina de atención a la interrupción del Timer0.
_tmr0_int:
```

```
push r16
ldi r16,0
mov r16, checksum_low
```

```
andi r16, 0x0F
ori r16,0b00010000
rcall sacanum
mov r16,checksum_high
andi r16, 0x0F
swap r16
ori r16,0b00100000
rcall sacanum
mov r16, checksum_high
andi r16, 0x0F
ori r16,0b01000000
rcall sacanum
```

```
mov r16,checksum_high
andi r16, 0b11110000
swap r16
ori r16,0b10000000
rcall sacanum
```

```
pop r16
reti
```

```
; ----------------------------------------------------------
; Rutina de atención a la interrupción por cambio en el estad
; ----------------------------------------------------------
; recordar que se configuró la detección por cambio para que
; se dispara la interrupción. LA interrupción no distingué qu
; Los botones se encuentran en PC.1, PC.2, PC.3 y recordar de
;
_pcint1:
sbis PINC,  1 ;si el boton1 esta presionado
inc r26
sbis PINC, 2 ;sino, si boton2 esta presionado
inc r26
sbis PINC, 3 ;sino, si boton 3 esta presionado
```

```
inc r26
reti
```

EXPLICACION DEL CODIGO

Inicio del Código Principal

```
.ORG 0x0000
jmp start         ; Dirección de comienzo (vector de reset)
.ORG 0x0008
jmp _pcint1       ; Salto a la rutina de atención a pcint1,
interrupción por cambio para los botones
.ORG 0x001C
jmp _tmr0_int     ; Salto atención a rutina de comparación
A del timer 0
```

Aquí se configuran los vectores de interrupción:

- `start`: La dirección de inicio del programa.
- `_pcint1`: Interrupción de los botones (cuando cambia el estado lógico de los botones).
- `_tmr0_int`: Interrupción del temporizador 0.

Segmento de Datos

```
.DSEG
buffer_msg: .byte 512         ; Reserva 512 bytes para el b
uffer de números aleatorios a transmitir
bmsg_end:   .byte 1           ; Marca el final del buffer
```

Este segmento en la memoria RAM define un espacio de 512 bytes para almacenar los números pseudoaleatorios y un byte para indicar el final del buffer (`bmsg_end`).

Inicio del Programa

```
.CSEG
start:
    call system_init
    ldi r26, 0x00             ; Bandera teclado
    mov checksum_high, r26    ; Inicializa el checksum a 0
    mov checksum_low, r26
    sei                       ; Habilita interrupciones par
a display y botones
    jmp modo_receptor
```

- `.CSEG` es una directiva en ensamblador que indica que lo que sigue es la **sección de código** del programa. Es decir, todo lo que esté después de `.CSEG` son las instrucciones del programa que el microcontrolador va a ejecutar, y se guardan en la **memoria de programa** (o memoria flash).

El programa comienza llamando a `system_init`, que inicializa los puertos, configuraciones de interrupciones, y el USART(es un periférico que permite a un microcontrolador enviar y recibir datos seriales). Luego, configura los registros `checksum_high` y `checksum_low` en cero (esto servirá para almacenar el checksum calculado) y habilita las interrupciones globales.

Después, el código salta a `modo_receptor`, lo que indica que el programa comienza en modo receptor de datos.

Modo Receptor

```
modo_receptor:
    ldi r26, 0
wait_4RX:                           ; Espera a que alguien
presione un botón
    sbrs r26, 0                     ; La interrupción del
botón cambia r26-bit0 a 1
    rjmp wait_4RX

    ; Ahora recibe 512 bytes y los deja en buffer_msg
    lds r16, UDR0                   ; Asegura que el buffe
```

```
r esté vacío
    lds r16, UDR0
    lds r16, UDR0
    cli                             ; Deshabilita interrup
ciones para display y botones
    rcall RX_512                    ; Llama a la subrutina
RX_512 para recibir 512 bytes
    sei                             ; Habilita interrupcio
nes nuevamente
    rcall Chksum_512                ; Calcula el nuevo che
cksum
    rjmp modo_receptor
```

En `modo_receptor`, el programa espera que se presione un botón (indicador en `r26`). Cuando detecta que un botón ha sido presionado, deshabilita las interrupciones y llama a `RX_512`, que recibe 512 bytes mediante USART y los almacena en `buffer_msg`. Luego, llama a `Chksum_512` para calcular el checksum de los datos recibidos. Finalmente, vuelve a esperar en `modo_receptor`.

1. Inicialización de `r26` como una bandera

```
assembly
Copiar código
ldi r26, 0
```

- `ldi r26, 0` carga el valor `0` en el registro `r26`.
- Aquí, `r26` se usa como una **bandera** para indicar si se ha presionado un botón.
- Esta bandera es modificada por la interrupción `_pcint1`, que se activa cuando se presiona un botón. Si se presiona, el bit 0 de `r26` se pone en `1`.

---

2. Esperar a que se presione un botón

```
assembly
Copiar código
```

```
wait_4RX:
    sbrs r26, 0
    rjmp wait_4RX
```

- Aquí tenemos un bucle de espera (`wait_4RX`).
- `sbrs r26, 0` (Skip if Bit in Register is Set) salta la siguiente instrucción si el bit 0 de `r26` es **1**.
- Si el bit 0 es **0**, entonces no salta y ejecuta `rjmp wait_4RX`, regresando al inicio del bucle.
- En resumen: este bucle espera hasta que el bit 0 de `r26` se ponga en `1`, lo cual indica que se presionó un botón (la interrupción `_pcint1` cambiaría este bit).

---

3. Descartar datos residuales en el buffer de recepción (`UDR0`)

```
assembly
Copiar código
lds r16, UDR0
lds r16, UDR0
lds r16, UDR0
```

- `lds r16, UDR0` carga el valor del registro `UDR0` en `r16`.
- `UDR0` es el registro de datos del USART (el puerto serie). Cuando recibimos datos por el USART, se almacenan aquí.
- Estas tres lecturas a `UDR0` son para asegurarse de que el buffer de recepción esté vacío y no contenga datos residuales antes de empezar a recibir los nuevos datos. Así evitamos procesar bytes "viejos".

---

4. Deshabilitar interrupciones antes de recibir los datos

```
assembly
Copiar código
```

```
cli
```

- `cli` (Clear Global Interrupt Flag) deshabilita todas las interrupciones globalmente.
- Esto se hace para asegurar que el proceso de recepción de datos no sea interrumpido mientras se reciben los 512 bytes.

---

5. Llamar a la subrutina `RX_512` para recibir 512 bytes

```
assembly
Copiar código
rcall RX_512
```

- `rcall RX_512` llama a la subrutina `RX_512`, que se encarga de recibir 512 bytes por el puerto serie (USART) y almacenarlos en el buffer `buffer_msg`.
- Esta subrutina usa "polling" para recibir cada byte y almacenarlo en memoria, un método algo ineficiente, pero común en este tipo de código de bajo nivel.
- **Polling** es una técnica para recibir datos donde el microcontrolador revisa repetidamente el estado de un registro (en este caso, el registro de estado del USART) para ver si hay datos nuevos listos para ser leídos, **en lugar de usar interrupciones**. En otras palabras, el microcontrolador "pregunta" continuamente si hay un nuevo byte disponible y, cuando lo hay, lo lee y lo almacena.

---

6. Rehabilitar las interrupciones después de recibir los datos

```
assembly
Copiar código
sei
```

- `sei` (Set Global Interrupt Flag) habilita las interrupciones nuevamente.
- Esto permite que el microcontrolador pueda responder a otras interrupciones (como las de botones o temporizador) después de completar la recepción de datos.

---

7. Calcular el Checksum de los datos recibidos

```
assembly
Copiar código
rcall Chksum_512
```

- `rcall Chksum_512` llama a la subrutina `Chksum_512`.
- Esta subrutina calcula el **checksum** (suma de comprobación) de los 512 bytes almacenados en `buffer_msg`. El resultado se guarda en los registros `checksum_high` y `checksum_low`.
- Este checksum puede utilizarse para verificar que los datos recibidos no hayan sufrido errores de transmisión.

---

8. Repetir el proceso de recepción (bucle infinito)

```
assembly
Copiar código
rjmp modo_receptor
```

- `rjmp modo_receptor` hace un salto sin condiciones al inicio de `modo_receptor`.
- Esto crea un **bucle infinito**, donde el microcontrolador espera constantemente a que se presione un botón, recibe 512 bytes, calcula el checksum, y vuelve a esperar.

`Chksum_512` - Cálculo del Checksum

```
assembly
Copiar código
Chksum_512:
```

```
    ldi YL, low(buffer_msg)         ; Apunta Y al primer byt
e del mensaje
    ldi YH, high(buffer_msg)
    clr checksum_high               ; Inicializa el checksum
    clr checksum_low

chksum_loop:
    ld datoTransmitido, Y+          ; Trae 1 byte a sumar
    add checksum_low, datoTransmitido   ; Suma el byte al c
hecksum (parte baja)
    adc checksum_high, temp_reg     ; Propaga el acarreo a l
a parte alta del checksum
    cpi YL, low(bmsg_end)           ; Comprueba si llegó al
final del buffer
    brne chksum_loop
    cpi YH, high(bmsg_end)
    brne chksum_loop
    ret
```

Esta subrutina calcula el checksum de los 512 bytes en `buffer_msg`. Utiliza el puntero Y (`YL` y `YH`) para recorrer cada byte en `buffer_msg`, y acumula el valor de cada byte en `checksum_low` y `checksum_high`. La operación `add` suma los bytes a `checksum_low`, y `adc` suma el acarreo a `checksum_high` en caso de que haya un overflow en `checksum_low`.

1. Inicializar el Puntero Y para Apuntar al Inicio del Buffer

```
assembly
Copiar código
ldi YL, low(buffer_msg)         ; Apunta Y al primer byte de
l mensaje
ldi YH, high(buffer_msg)
```

- `ldi YL, low(buffer_msg)` y `ldi YH, high(buffer_msg)` cargan la dirección de inicio de `buffer_msg` en el puntero `Y` (que consiste en los registros `YL` y `YH`).

- El puntero **Y** es un registro especial de 16 bits que se usa para apuntar a direcciones en la memoria. Los registros en los microcontroladores AVR tienen un tamaño de 8 bits, por lo que para representar una dirección de 16 bits (que es la longitud típica de una dirección de memoria), el puntero Y se divide en dos registros de 8 bits:
- `YL`: Es el registro bajo de Y y contiene los 8 bits menos significativos de la dirección.
- `YH`: Es el registro alto de Y y contiene los 8 bits más significativos de la dirección.

Entonces, cuando hablamos de `Y`, nos referimos al conjunto de `YH:YL`, que juntos forman un puntero de 16 bits.

¿Cuáles son los registros físicos que representan a YL y YH?

En los microcontroladores AVR:

- `YH` es en realidad el registro `r29`.
- `YL` es el registro `r28`.

Por lo tanto:

- `Y = YH:YL = r29:r28`
- `buffer_msg` es el buffer en memoria RAM donde están almacenados los 512 bytes de datos que vamos a procesar.
- Al cargar la dirección de `buffer_msg` en `Y`, estamos listos para recorrer el buffer y acceder a cada byte.

---

2. Inicializar el Checksum en Cero

```
assembly
Copiar código
clr checksum_high              ; Inicializa el checksum
clr checksum_low
```

- `clr checksum_high` y `clr checksum_low` ponen a cero los registros `checksum_high` y `checksum_low`, respectivamente.

- Estos registros guardarán el resultado del checksum de 16 bits (la suma de todos los bytes en el buffer).
  - `checksum_low` almacena la parte baja (8 bits) del checksum.
  - `checksum_high` almacena la parte alta (8 bits) del checksum.
- Al inicializar ambos registros en cero, nos aseguramos de que el cálculo comience desde un valor inicial neutro.

---

3. Bucle para Sumar Cada Byte del Buffer

```
assembly
Copiar código
chksum_loop:
    ld datoTransmitido, Y+          ; Trae 1 byte a sumar
    add checksum_low, datoTransmitido   ; Suma el byte al c
hecksum (parte baja)
    adc checksum_high, temp_reg     ; Propaga el acarreo a l
a parte alta del checksum
```

Explicación Paso a Paso:

1. **Cargar un Byte desde el Buffer**:

   ```
   assembly
   Copiar código
   ld datoTransmitido, Y+
   ```

   - `ld datoTransmitido, Y+` carga el byte apuntado por `Y` en el registro `datoTransmitido` y luego incrementa el puntero `Y` para apuntar al siguiente byte.
   - Esto permite recorrer el buffer `buffer_msg` byte por byte, leyendo un valor diferente en cada iteración del bucle.

2. **Suma del Byte al Checksum (Parte Baja)**:

```
add checksum_low, datoTransmitido
```

- `add checksum_low, datoTransmitido` suma el valor de `datoTransmitido` a `checksum_low`.
- Esta es una operación de 8 bits, por lo que si el resultado sobrepasa 255 (es decir, hay un "carry" o acarreo), el carry se propagará a `checksum_high` en el siguiente paso.

3. **Propagación del Carry a la Parte Alta del Checksum**:

```
assembly
Copiar código
adc checksum_high, temp_reg
```

- `adc` significa "Add with Carry" (sumar con acarreo).
- En este caso, estamos usando `temp_reg`, que es un registro que ya está en cero (generalmente `temp_reg` es `r1`, que suele estar en cero en muchos códigos AVR).
- `adc checksum_high, temp_reg` suma el carry resultante de la operación anterior (si lo hubo) a `checksum_high`.
- Esto asegura que si hubo un overflow en `checksum_low`, se refleje correctamente en `checksum_high`, manteniendo el valor de checksum como un número de 16 bits.
- `temp_reg` es una etiqueta para el registro `r1`, que en este código se usa como un registro temporal.
- Se utiliza en `Chksum_512` para propagar el acarreo (carry) al hacer la suma del checksum.
- En el contexto del cálculo del checksum, `temp_reg` **siempre tiene un valor de cero**, permitiendo que `adc` solo sume el acarreo sin modificar el valor original de `checksum_high`.

---

4. Verificar si se Ha Llegado al Final del Buffer

```
assembly
Copiar código
    cpi YL, low(bmsg_end)          ; Comprueba si llegó al
final del buffer
    brne chksum_loop
    cpi YH, high(bmsg_end)
    brne chksum_loop
```

- `cpi YL, low(bmsg_end)` y `cpi YH, high(bmsg_end)` comparan el valor del puntero `Y` con la dirección de `bmsg_end`, que marca el final del buffer.
- `brne chksum_loop` (Branch if Not Equal) verifica si la comparación fue desigual. Si el puntero `Y` no ha llegado al final del buffer, el programa vuelve a `chksum_loop` para procesar el siguiente byte.
- Una vez que `Y` alcanza la dirección de `bmsg_end`, el bucle termina y la subrutina sigue a la siguiente instrucción (`ret`), finalizando el cálculo del checksum.

---

5. Retornar

```
assembly
Copiar código
ret
```

- `ret` indica el fin de la subrutina `Chksum_512`.
- Al retornar, los valores de `checksum_high` y `checksum_low` contienen el checksum calculado para los 512 bytes en `buffer_msg`.

```
; TX - rutina de transmisión serial USART. Transmite los 512
;-------------------------------------------------------------
;TX_512:
;   ldi ZL, low(buffer_msg)        ; Apunto Z al primer byte
;  ldi ZH, high(buffer_msg)
```

```
; ldi r16, (1 << TXEN0)          ; Habilito el transmisor USA
;sts UCSR0B, r16                ;carga el valor de r16 en UCS
```

```
;TX_loop1:
;   ld datoTransmitido, Z+        ; Traigo el Byte a transmi
;  sts UDR0, datoTransmitido      ; almacenar el contenido de
```

```
;TX_loop2:
;   lds r16, UCSR0A               ; Espero a que termine la
;  sbrs r16, UDRE0
; rjmp TX_loop2
```

```
;cpi ZL, low(bmsg_end)          ; Chequeo si llegué al fina
l del buffer
;brne TX_loop1
;cpi ZH, high(bmsg_end)
;brne TX_loop1

;ret
```

```
;chequeo si llegué al final del buffer
```

```
;ret
```

1. `TX_512:` define el inicio de la subrutina.
2. `ldi ZL, low(buffer_msg)` y `ldi ZH, high(buffer_msg)` cargan el registro de 16 bits `Z` con la dirección de inicio de `buffer_msg`. Esto permite que `Z` apunte al primer byte del mensaje almacenado en `buffer_msg`.

```
assembly
Copiar código
    ldi r16, (1 << TXEN0)          ; Habilito el transmisor
USART
```

```
    sts UCSR0B, r16
```

1. `ldi r16, (1 << TXEN0)` carga en el registro `r16` un valor que habilita el transmisor USART. El bit `TXEN0` es el que habilita la transmisión de datos.
2. `sts UCSR0B, r16` almacena el valor en `UCSR0B`, que es el registro de control del USART. Al hacer esto, el transmisor USART se habilita y queda listo para enviar datos.

La línea

assembly
Copiar código
sts UDR0, datoTransmitido

transmitir un byte de datos a través de la interfaz USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter) en un microcontrolador AVR.

Desglose de la operación:

1. `sts` **(Store Direct to SRAM/IO Register)**:
   - Es una instrucción AVR que almacena el contenido de un registro general en una ubicación específica de la memoria de datos o en un registro de I/O.
   - En este caso, almacena el valor del registro `datoTransmitido` en la dirección del registro `UDR0`.
2. `UDR0` **(USART Data Register 0)**:
   - Este registro es el *buffer* de datos para el módulo USART del microcontrolador.
   - Es el punto donde se escriben los datos que se desean transmitir por el puerto serie.
   - Cuando se escribe un valor en `UDR0`, el hardware de USART automáticamente inicia la transmisión del byte a través de la línea de transmisión (TX).

3. `datoTransmitido`:
    - Es un registro del microcontrolador (definido como `r0` en tu código).
    - Contiene el byte que deseas transmitir.

Lo que hace la línea:

1. El contenido del registro `datoTransmitido` se copia al registro `UDR0`.
2. El hardware USART toma ese byte y lo coloca en el *buffer de transmisión*, comenzando la transmisión automáticamente.
3. Dependiendo de la configuración de USART (velocidad en baudios, paridad, bits de datos, etc.), el byte se enviará por la línea TX en el formato configurado.

Ejemplo práctico:

Supongamos que el valor en `datoTransmitido` es `0x41` (el carácter `A` en ASCII). Cuando ejecutas:

```
assembly
Copiar código
sts UDR0, datoTransmitido
```

- El valor `0x41` se coloca en `UDR0`.
- La interfaz USART envía este byte, que se interpretará como el carácter `A` en el dispositivo receptor (como un terminal serie en una PC).

UCSR0A (USART Control and Status Register 0 A): Este registro contiene varios bits de estado y control relacionados con el funcionamiento del USART. Uno de estos bits es UDRE0, que se utiliza para indicar si el registro de datos de transmisión (UDR0) está listo para aceptar un nuevo byte de datos para transmitir.

UDRE0 (USART Data Register Empty): Este bit específico en el registro UCSR0A indica si el registro de datos (UDR0) está vacío y listo para recibir un nuevo byte. Cuando UDRE0 está en 1, significa que el dato actual ha sido transmitido y el UDR0 está disponible para cargar el siguiente byte. En cambio, cuando

está en 0, indica que la transmisión del byte aún está en curso y que UDR0 no está listo para recibir datos nuevos.

Siguiente bloque

Después de `TX_loop2`, el código verifica si ha alcanzado el final del buffer de datos que se está transmitiendo.

```
assembly
Copiar código
    cpi ZL, low(bmsg_end)          ; Compara ZL con la part
e baja de la dirección de fin del buffer
    brne TX_loop1                  ; Si ZL no coincide con
el final, salta a TX_loop1
    cpi ZH, high(bmsg_end)         ; Compara ZH con la part
e alta de la dirección de fin del buffer
    brne TX_loop1                  ; Si ZH no coincide con
el final, salta a TX_loop1
```

1. **`cpi ZL, low(bmsg_end)`**: Compara el registro `ZL` (parte baja del puntero `Z`) con el valor de `low(bmsg_end)`, que es la parte baja de la dirección de fin del buffer (`bmsg_end`). Esto es para verificar si se ha alcanzado el final del buffer.
2. **`brne TX_loop1`**: Si `ZL` no coincide con `low(bmsg_end)`, esto significa que **aún no hemos llegado al final del buffer**, por lo que el código salta a `TX_loop1` para continuar transmitiendo más datos.
3. **`cpi ZH, high(bmsg_end)`**: Si `ZL` coincide con `low(bmsg_end)`, entonces se compara `ZH` (parte alta de `Z`) con `high(bmsg_end)`, que es la parte alta de la dirección de `bmsg_end`. Si también coinciden, significa que hemos llegado al final del buffer.
4. **`brne TX_loop1`**: Si `ZH` no coincide con `high(bmsg_end)`, se salta nuevamente a `TX_loop1`, ya que todavía no se ha alcanzado el final del buffer completo.

`ret`

Finalmente, si ambas comparaciones (`ZL` y `ZH`) coinciden con `bmsg_end`, el bucle termina y el código continúa con la instrucción `ret` para salir de la subrutina de transmisión.

`RX_512` - Recepción de Datos

```
assembly
Copiar código
RX_512:
    ldi ZL, low(buffer_msg)        ; Apunta Z al primer byt
e del buffer
    ldi ZH, high(buffer_msg)
    ldi r16, (1 << RXEN0)          ; Configura el USART com
o receptor
    sts UCSR0B, r16

RX_Wait:
    lds r16, UCSR0A                ; Polling para verificar
recepción de datos
    sbrs r16, RXC0
    rjmp RX_Wait

    lds datoTransmitido, UDR0      ; Almacena el byte recib
ido
    st Z+, datoTransmitido         ; Guarda el byte en el b
uffer

    cpi ZL, low(bmsg_end)          ; Comprueba si llegó al
final del buffer
    brne RX_Wait
    cpi ZH, high(bmsg_end)
    brne RX_Wait

    ret
```

`RX_512` recibe 512 bytes de datos por USART. Utiliza el registro Z (`ZL` y `ZH`) para apuntar al buffer `buffer_msg`, y realiza un polling en `UCSR0A` para esperar que el dato esté listo en el registro `UDR0`. Los bytes recibidos se almacenan secuencialmente en `buffer_msg`.

`RX_512` : Configuración inicial para recibir 512 bytes

La subrutina `RX_512` tiene como objetivo recibir 512 bytes por USART (puerto serie) y almacenarlos en `buffer_msg`. Los primeros pasos son inicializar el puntero `Z` y configurar el USART en modo receptor

```
RX_512:
    ldi ZL, low(buffer_msg)         ; Apunta Z al primer byt
e del buffer
    ldi ZH, high(buffer_msg)
```

- Aquí, `Z` es un puntero de 16 bits que se compone de los registros `ZL` y `ZH`.
- `ldi ZL, low(buffer_msg)` y `ldi ZH, high(buffer_msg)` cargan la dirección de inicio de `buffer_msg` en el puntero `Z`, para que podamos usarlo para almacenar los bytes que se van recibiendo.
- `ZL`: Se refiere a `R30`, la parte baja del puntero `Z`.
- `ZH`: Se refiere a `R31`, la parte alta del puntero `Z`.

```
assembly
Copiar código
    ldi r16, (1 << RXEN0)           ; Configura el USART com
o receptor
    sts UCSR0B, r16
```

- `ldi r16, (1 << RXEN0)` carga en `r16` el valor `(1 << RXEN0)`, que es un valor que activa el bit `RXEN0` en el registro `UCSR0B`.
- `RXEN0` es el bit que habilita el receptor del USART, lo que permite al microcontrolador recibir datos en el puerto serie.
- `sts UCSR0B, r16` guarda este valor en el registro `UCSR0B`, activando así el receptor.

2. `RX_Wait` : Esperar a que llegue un byte (Polling)

```
assembly
Copiar código
RX_Wait:
    lds r16, UCSR0A                ; Ahora polling para esp
erar recibir algo (UDR0)
    sbrs r16, RXC0
    rjmp RX_Wait
```

Esta sección de código (`RX_Wait`) realiza **polling** para esperar a que llegue un byte de datos en el puerto serie. Vamos a ver cómo funciona cada instrucción:

- `lds r16, UCSR0A` carga el valor del registro `UCSR0A` en `r16`. `UCSR0A` es el registro de estado del USART y contiene un bit llamado `RXC0` (Receive Complete) que indica si se ha recibido un byte.
- `sbrs r16, RXC0` verifica el bit `RXC0` en `r16`.
  - `sbrs` significa "Skip if Bit in Register is Set" (salta si el bit en el registro está activado).
  - Si `RXC0` es 0 (es decir, **no se ha recibido ningún byte todavía**), `sbrs` no salta y ejecuta la siguiente instrucción (`rjmp RX_Wait`), que vuelve al inicio de `RX_Wait`.
  - Si `RXC0` es 1 (**hay un byte disponible en el registro de datos** `UDR0`), `sbrs` salta la siguiente instrucción y continúa ejecutando el código que sigue.
  - **`lds datoTransmitido, UDR0`**
    - Una vez que hay un dato disponible (es decir, `RXC0` está en alto), este se carga en `datoTransmitido` desde el registro `UDR0`.
  - **`st Z+, datoTransmitido`**
    - Guarda el dato recibido (`datoTransmitido`) en la posición de memoria apuntada por el puntero `Z` (que aquí está apuntando a `buffer_msg`) y luego incrementa `Z` para apuntar al siguiente espacio en el buffer.
    - La instrucción `st` en el ensamblador AVR se usa para **almacenar** un dato en la memoria a la que apunta un registro de puntero (como `X`, `Y` o `Z`).

- `cpi ZL, low(bmsg_end)` y `brne RX_Wait`
  - Verifica si `Z` ha alcanzado el final del buffer (`bmsg_end`). Si `ZL` aún no alcanza el límite bajo (`low(bmsg_end)`), salta de vuelta a `RX_Wait` para recibir el siguiente byte.
- `cpi ZH, high(bmsg_end)` y `brne RX_Wait`
  - Si `ZL` llegó al límite bajo, ahora verifica la parte alta `ZH` para asegurarse de que `Z` completo llegó al final del buffer.

En resumen, este bucle de polling (`RX_Wait`) mantiene al microcontrolador en espera hasta que se reciba un byte. Una vez que `RXC0` se activa, el programa sale del bucle y continúa.

3. Leer y almacenar el byte recibido

```
assembly
Copiar código
    lds datoTransmitido, UDR0      ; Llego aquí solo si rec
ibí algo
    st Z+, datoTransmitido         ; Guardo lo que recibí
```

- `lds datoTransmitido, UDR0` lee el byte recibido desde el registro `UDR0` y lo guarda en `datoTransmitido`.
  - `UDR0` es el registro de datos del USART, donde se almacenan los bytes recibidos.
- `st Z+, datoTransmitido` almacena el valor de `datoTransmitido` en la dirección apuntada por `Z` (inicialmente apuntando a `buffer_msg`).
  - La sintaxis `Z+` significa que después de almacenar el dato, **incrementa el puntero** `Z` para que apunte a la siguiente posición en el buffer. Esto permite que cada byte recibido se almacene en una posición consecutiva en `buffer_msg`.

4. Verificar si hemos llegado al final del buffer

```
assembly
Copiar código
    cpi ZL, low(bmsg_end)          ; Chequeo si llegué al f
```

```
inal del buffer
    brne RX_Wait
    cpi ZH, high(bmsg_end)
    brne RX_Wait
```

- `cpi ZL, low(bmsg_end)` y `cpi ZH, high(bmsg_end)` comparan el valor actual del puntero `Z` con la dirección de `bmsg_end`, que marca el final del buffer `buffer_msg`.
    - `cpi` significa "Compare with Immediate" (compara con un valor inmediato).
- `brne RX_Wait` significa "Branch if Not Equal" (ramifica si no son iguales). Si `ZL` o `ZH` no son iguales a `low(bmsg_end)` o `high(bmsg_end)`, respectivamente, eso significa que aún no hemos llegado al final del buffer, por lo que el código vuelve a `RX_Wait` para esperar el siguiente byte.

Este bucle continúa hasta que `Z` alcanza el valor de `bmsg_end`, lo cual indica que hemos recibido todos los bytes requeridos (en este caso, 512 bytes).

5. Finalizar la subrutina

```
assembly
Copiar código
    ret
```

- `ret` marca el final de la subrutina `RX_512`.
- Cuando el código llega a esta instrucción, significa que se han recibido los 512 bytes y se han almacenado en `buffer_msg`.

Generación de Números Pseudoaleatorios con XORSHIFT

```
assembly
Copiar código
aleatorios:
    ldi r28, low(buffer_msg)          ; Apunta Y al primer b
yte del buffer
```

```
    ldi r29, high(buffer_msg)
ale_loop:
    ldi r20, 13
    call ale_loop_l
    ldi r20, 17
    call ale_loop_r
    ldi r20, 5
    call ale_loop_l

    st Y+, r16                          ; Guarda los 32 bits
generados en el buffer
    st Y+, r17
    st Y+, r18
    st Y+, r19
    cpi YL, low(bmsg_end)
    brne ale_loop
    cpi YH, high(bmsg_end)
    brne ale_loop
    ret
```

La rutina `aleatorios` genera 512 bytes de números pseudoaleatorios y los almacena en `buffer_msg` usando un algoritmo XORSHIFT. Utiliza `ale_loop_l` y `ale_loop_r` para rotar y mezclar los registros `r16`, `r17`, `r18` y `r19` en base al valor en `r20`, generando un nuevo número de 32 bits en cada iteración y almacenándolo en el buffer.

1. Inicialización del Puntero Y para Apuntar al Buffer

```
assembly
Copiar código
ldi     r28,    low(buffer_msg)     ; Apunta Y al primer by
te del buffer
ldi     r29,    high(buffer_msg)
```

- `ldi r28, low(buffer_msg)` y `ldi r29, high(buffer_msg)` cargan la dirección de inicio de `buffer_msg` en el puntero `Y` (compuesto por `r28` y `r29`).

- Esto permite que `Y` apunte al primer byte del buffer `buffer_msg`, donde se almacenarán los números pseudoaleatorios generados.

2. Generación de un Número Aleatorio de 32 Bits usando XORSHIFT

El algoritmo XORSHIFT se implementa aquí mediante una serie de desplazamientos y operaciones XOR en los registros `r16`, `r17`, `r18`, y `r19`, que juntos representan un número de 32 bits.

```
assembly
Copiar código
ldi     r20,    13
call    ale_loop_l
ldi     r20,    17
call    ale_loop_r
ldi     r20,    5
call    ale_loop_l
```

Cada una de estas llamadas hace lo siguiente:

- `ldi r20, 13` **seguido de** `call ale_loop_l`: realiza un desplazamiento de 13 posiciones a la izquierda en los registros `r16:r17:r18:r19`.
- **Cargar r20 y llamar a la subrutina**:
  - La instrucción `ldi r20, 13` carga el número `13` en `r20`, y luego se llama a `ale_loop_l`.
  - Dentro de `ale_loop_l`, se realizan 13 rotaciones de bits a la izquierda en los registros `r16` a `r19`.
- **Rotaciones a la Izquierda (`ale_loop_l`) y Derecha (`ale_loop_r`)**:
  - La subrutina `ale_loop_l` hace rotaciones de bits a la izquierda, mientras que `ale_loop_r` hace rotaciones a la derecha.
  - El número de rotaciones está determinado por el valor que se carga en `r20`. En este caso, `13` rotaciones a la izquierda, `17` a la derecha y luego `5` más a la izquierda.
- **Generación del Número Pseudoaleatorio**:

    - Después de cada conjunto de rotaciones, se realiza una operación XOR entre los valores en los registros `r16` a `r19`.
    - Este proceso de rotaciones y XOR genera un nuevo valor en los registros `r16` a `r19`, que se considera un número pseudoaleatorio.

- `ldi r20, 17` **seguido de** `call ale_loop_r`: realiza un desplazamiento de 17 posiciones a la derecha en los registros `r16:r17:r18:r19`.
- `ldi r20, 5` **seguido de** `call ale_loop_l`: realiza un desplazamiento de 5 posiciones a la izquierda en los registros `r16:r17:r18:r19`.

Este proceso de **desplazamiento** y **XOR** crea un número pseudoaleatorio cada vez que se ejecuta, con base en la "semilla" inicial que estaba en `r16:r17:r18:r19`.

Vamos a ver en detalle cómo funcionan `ale_loop_l` y `ale_loop_r`.

`ale_loop_l`: Desplazamiento a la Izquierda con XOR

```
assembly
Copiar código
ale_loop_l:
    mov     datoTransmitido,    r16
    mov     r1,     r17
    mov     r2,     r18
    mov     r3,     r19
ale_rota_l:
    clc
    rol     datoTransmitido
    rol     r1
    rol     r2
    rol     r3
    dec     r20
    brne    ale_rota_l
ale_rota_out:
    eor     r16,    datoTransmitido
    eor     r17,    r1
    eor     r18,    r2
    eor     r19,    r3
```

```
    ret
```

Explicación de `ale_loop_l`:

1. **Inicialización**:
   - `mov datoTransmitido, r16`, `mov r1, r17`, `mov r2, r18`, y `mov r3, r19` copian los valores actuales de `r16:r17:r18:r19` en registros temporales (`datoTransmitido`, `r1`, `r2`, y `r3`). Esto permite realizar operaciones de desplazamiento sin perder los valores originales.
2. **Bucle de Desplazamiento a la Izquierda**:
   - `rol` (Rotate Left) desplaza cada registro a la izquierda (con acarreo). Esto mueve los bits un lugar hacia la izquierda.
   - Este bucle se repite `r20` veces (en este caso, `r20` es 13 o 5, según el valor cargado antes de llamar a `ale_loop_l`).
   - `dec r20` decrementa `r20` en cada iteración, y `brne ale_rota_l` vuelve al inicio del bucle si `r20` no ha llegado a cero.
3. **Operación XOR para Crear el Número Pseudoaleatorio**:
   - Después del desplazamiento, se aplica `eor` (XOR) entre cada par de registros originales y sus valores desplazados.
   - `eor r16, datoTransmitido`, `eor r17, r1`, `eor r18, r2`, y `eor r19, r3` generan un nuevo número de 32 bits en `r16:r17:r18:r19` al mezclar (XOR) los valores desplazados con los originales.

`ale_loop_r`: Desplazamiento a la Derecha con XOR

```
assembly
Copiar código
ale_loop_r:
    mov     datoTransmitido,    r16
    mov     r1,     r17
    mov     r2,     r18
    mov     r3,     r19
ale_rota_r:
    clc
```

```
    ror     r3
    ror     r2
    ror     r1
    ror     datoTransmitido
    dec     r20
    brne    ale_rota_r
ale_rota_out:
    eor     r16,    datoTransmitido
    eor     r17,    r1
    eor     r18,    r2
    eor     r19,    r3
    ret
```

Explicación de `ale_loop_r`:

La estructura es muy similar a `ale_loop_l`, pero en este caso se realiza un **desplazamiento a la derecha**.

1. **Inicialización**:
   - `mov datoTransmitido, r16`, `mov r1, r17`, `mov r2, r18`, y `mov r3, r19` copian los valores originales de `r16:r17:r18:r19` en registros temporales.
2. **Bucle de Desplazamiento a la Derecha**:
   - `ror` (Rotate Right) desplaza cada registro un bit a la derecha (con acarreo), moviendo los bits hacia la derecha.
   - Este bucle se repite `r20` veces (en este caso, `r20` es 17 según el valor cargado antes de llamar a `ale_loop_r`).
3. **Operación XOR para Crear el Número Pseudoaleatorio**:
   - Después del desplazamiento, `eor` se utiliza para combinar los valores desplazados con los valores originales, generando un nuevo número pseudoaleatorio de 32 bits en `r16:r17:r18:r19`.

---

3. Guardar los 32 Bits en el Buffer

```
assembly
Copiar código
```

```
st Y+, r16        ; Guarda el byte bajo del número pseudoal
eatorio
st Y+, r17
st Y+, r18
st Y+, r19
```

- `st Y+, r16` almacena el valor de `r16` en la dirección apuntada por `Y` y luego incrementa el puntero `Y`.
- Esto se repite para `r17`, `r18`, y `r19`, almacenando los 4 bytes (32 bits) en el buffer `buffer_msg`.
- `Y` se incrementa automáticamente después de cada instrucción `st Y+`, de modo que el siguiente número pseudoaleatorio se almacenará en la posición correcta en el buffer.

---

4. Verificar si Hemos Llegado al Final del Buffer

```
assembly
Copiar código
cpi YL, low(bmsg_end)
brne ale_loop
cpi YH, high(bmsg_end)
brne ale_loop
```

- `cpi YL, low(bmsg_end)` y `cpi YH, high(bmsg_end)` comparan el valor actual del puntero `Y` con `bmsg_end`, que marca el final de `buffer_msg`.
- Si no hemos llegado al final, `brne ale_loop` regresa a `ale_loop` para generar el siguiente número pseudoaleatorio.
- Cuando `Y` alcanza `bmsg_end`, se detiene el bucle y la rutina termina con `ret`.

```
sacanum:
push r16                ; guardo una copia de r16
ldi zh, high(segmap<<1) ; Initialize Z-pointer
ldi zl, low(segmap<<1)
andi r16, 0x0F
```

```
add zl, r16
clr r16
adc zh, r16
lpm r16, Z              ; traigo de la memoria de Programa el
call sacabyte
pop r16
call sacabyte
sbi PORTD, 4            ; PD.4 a 1, es el reloj del latch
cbi PORTD, 4            ; PD.4 a 0, es el reloj del latch
ret
```

```
;-----------------------------------------------------------
;   SACABYTE
;-----------------------------------------------------------
; saca un byte por el 7seg
sacabyte:
ldi r17, 0x08
loop_byte1:
cbi PORTD, 7            ; SCLK = 0
lsr r16
brcs loop_byte2         ; salta si C=1
cbi PORTB, 0            ; SD = 0
rjmp loop_byte3
loop_byte2:
sbi PORTB, 0            ; SD = 1
loop_byte3:
sbi PORTD, 7            ; SCLK = 1
dec r17
brne loop_byte1
ret
```

Subrutina `sacanum`

Esta subrutina toma un valor de 4 bits en `r16` y lo convierte en un valor para el display de 7 segmentos usando una tabla de mapeo (almacenada en `segmap`). Luego, envía el valor resultante al display a través de la subrutina `sacabyte`. Veamos cada instrucción:

1. `push r16`: Guarda el valor original de `r16` en la pila para que esté disponible más adelante, ya que `r16` se usará en el resto de la subrutina.
2. `ldi zh, high(segmap<<1)` **y** `ldi zl, low(segmap<<1)`: Inicializa el puntero `Z` (`ZH:ZL`) con la dirección de inicio de `segmap` en la memoria de programa. Aquí, `segmap<<1` se usa porque la memoria de programa está organizada en palabras de 2 bytes, y se necesita el valor desplazado para acceder a cada byte en esa memoria.
3. `andi r16, 0x0F`: Asegura que `r16` tenga solo los 4 bits menos significativos (0-15), que es el rango válido para un dígito hexadecimal (0-F). Esto es útil si `r16` puede tener más de 4 bits y necesitamos solo el valor del dígito.
4. `add zl, r16`: Suma el valor en `r16` al registro `ZL`, para apuntar al valor correspondiente en la tabla `segmap`.
5. `clr r16`: Limpia `r16` (lo pone en 0) para preparar una suma con acarreo en la siguiente instrucción.
6. `adc zh, r16`: Suma con acarreo el contenido de `r16` (que ahora es 0) a `ZH`. Esto ajusta `ZH` en caso de que la suma anterior (`add zl, r16`) haya generado un acarreo.

Así es, la instrucción `adc zh, r16` **suma el valor de** `r16` **y el bit de acarreo (** `C` **) al registro** `ZH`. Pero, como `r16` fue previamente limpiado a 0 con `clr r16`, esta operación no afectará a `ZH` excepto si el bit de acarreo está en 1.

Para resumir cómo funciona:

- `adc zh, r16` suma el valor de `r16` (0) y el bit de acarreo (`C`) al registro `ZH`.
- Esto significa que:
  - **Si el bit de acarreo (** `C` **) es 0**, `adc zh, r16` no cambia el valor de `ZH`.
  - **Si el bit de acarreo (** `C` **) es 1**, `adc zh, r16` incrementa el valor de `ZH` en 1.

7. `lpm r16, Z`: Carga el valor apuntado por `Z` en `r16` desde la memoria de programa. Este valor corresponde al código de 7 segmentos para el dígito que estábamos buscando en `segmap`.
8. `call sacabyte`: Llama a la subrutina `sacabyte` para enviar el byte (`r16`) al display de 7 segmentos.
9. `pop r16`: Restaura el valor original de `r16` desde la pila.

10. `call sacabyte`: Vuelve a llamar a `sacabyte`, probablemente para enviar el mismo byte o algún valor de control a otra parte del sistema.
11. `sbi PORTD, 4` y `cbi PORTD, 4`: Configura el pin PD4 como un pulso de reloj para el latch (bloqueo). Al cambiar el estado de este pin (de 0 a 1 y luego de vuelta a 0), se asegura que el valor que se ha enviado esté registrado en el latch del display.
12. `ret`: Termina la subrutina `sacanum` y vuelve a la función que la llamó.

Subrutina `sacabyte`

Esta subrutina envía un byte (`r16`) al display de 7 segmentos bit a bit. Cada bit se envía en serie, comenzando por el bit menos significativo. Veamos cómo funciona:

1. `ldi r17, 0x08`: Carga el registro `r17` con el valor 8, que será usado como contador de bits. La subrutina va a enviar 8 bits (1 byte), uno por uno.
2. `loop_byte1:`: Etiqueta del inicio del bucle para enviar cada bit.
3. `cbi PORTD, 7`: Baja el reloj serial (`SCLK`) a 0, indicando el inicio de un ciclo de reloj.
4. `lsr r16`: Desplaza el contenido de `r16` un bit a la derecha, moviendo el bit menos significativo al bit de acarreo (`C`). Esto permite verificar si el bit menos significativo era 1 o 0.
5. `brcs loop_byte2`: Salta a `loop_byte2` si el bit desplazado era 1 (es decir, si el bit de acarreo está en 1).
6. `cbi PORTB, 0`: Si el bit desplazado era 0, se pone el pin `SD` en 0 (señal serial de datos).
7. `rjmp loop_byte3`: Salta a `loop_byte3`, evitando la siguiente instrucción si el bit era 0.
8. `loop_byte2:`: Etiqueta que se ejecuta solo si el bit desplazado era 1.
9. `sbi PORTB, 0`: Si el bit desplazado era 1, se pone el pin `SD` en 1 (señal serial de datos).
10. `loop_byte3:`: Continúa el flujo común después de haber configurado el bit en el pin `SD`.

11. `sbi PORTD, 7`: Sube el reloj serial (`SCLK`) a 1, lo que permite registrar el bit actual en el display.
12. `dec r17`: Decrementa el contador de bits (`r17`).
13. `brne loop_byte1`: Repite el bucle si aún quedan bits por enviar (si `r17` no es 0).
14. `ret`: Termina la subrutina `sacabyte` y regresa a la instrucción siguiente después de la llamada a `sacabyte`.

Interrupción de Timer (`_tmr0_int`)

```
assembly
Copiar código
_tmr0_int:
    push r16
    ldi r16,0
    mov r16, checksum_low
    andi r16, 0x0F
    ori r16,0b00010000
    rcall sacanum
    mov r16,checksum_high
    andi r16, 0x0F
    swap r16
    ori r16,0b00100000
    rcall sacanum
    mov r16, checksum_high
    andi r16, 0x0F
    ori r16,0b01000000
    rcall sacanum
    mov r16,checksum_high
    andi r16, 0b11110000
    swap r16
    ori r16,0b10000000
    rcall sacanum
    pop r16
```

```
    reti
```

Esta interrupción se dispara por el temporizador 0, y se encarga de mostrar el valor del checksum en el display de 7 segmentos. Los detalles de cómo se manipulan los datos en el display están en otras subrutinas como `sacanum`.

1. Guardar el Registro `r16` (Push)

```
assembly
Copiar código
push r16
```

- Esta línea guarda el valor actual de `r16` en la **pila** (stack) antes de modificarlo en la rutina.
- Como `r16` se va a usar dentro de esta interrupción para mostrar los valores en el display, guardarlo al inicio permite restaurarlo al final de la interrupción, asegurando que no se altere el valor original de `r16` al regresar a la rutina principal.

---

2. Preparar y Mostrar el Primer Dígito del Checksum

```
assembly
Copiar código
ldi r16, 0
mov r16, checksum_low
andi r16, 0x0F
ori r16, 0b00010000
rcall sacanum
```

Aquí se prepara el primer dígito a mostrar en el display:

- `mov r16, checksum_low` copia el valor de `checksum_low` (la parte baja del checksum de 16 bits) en `r16`.
- `andi r16, 0x0F` aplica una **máscara** para extraer solo los 4 bits menos significativos (o sea, el primer dígito hexadecimal) de `checksum_low`.

- `ori r16, 0b00010000` establece el bit de control para indicar que este es el primer dígito (probablemente representa cuál de los dígitos del display de 7 segmentos queremos activar).
- `rcall sacanum` llama a la subrutina `sacanum`, que se encarga de mostrar el valor de `r16` en el display de 7 segmentos.

Resultado: El primer dígito del `checksum_low` se muestra en el display.

---

3. Preparar y Mostrar el Segundo Dígito del Checksum

```
assembly
Copiar código
mov r16, checksum_high
andi r16, 0x0F
swap r16
ori r16, 0b00100000
rcall sacanum
```

Esta parte del código se encarga de mostrar el segundo dígito:

- `mov r16, checksum_high` copia el valor de `checksum_high` (la parte alta del checksum de 16 bits) en `r16`.
- `andi r16, 0x0F` aplica una **máscara** para extraer solo los 4 bits menos significativos de `checksum_high`.
- `swap r16` intercambia los 4 bits altos y bajos en `r16`. Esto coloca los bits menos significativos en la posición correcta para mostrarlos en el display.
- `ori r16, 0b00100000` establece un bit de control para indicar que estamos mostrando el segundo dígito.
- `rcall sacanum` llama a `sacanum` para mostrar el valor en el display.

Resultado: El segundo dígito del `checksum_high` se muestra en el display.

---

4. Preparar y Mostrar el Tercer Dígito del Checksum

```
assembly
Copiar código
mov r16, checksum_high
```

```
andi r16, 0x0F
ori r16, 0b01000000
rcall sacanum
```

Esta parte prepara el tercer dígito a mostrar:

- `mov r16, checksum_high` copia nuevamente el valor de `checksum_high` en `r16`.
- `andi r16, 0x0F` aplica una **máscara** para extraer solo los 4 bits menos significativos de `checksum_high`.
- `ori r16, 0b01000000` configura un bit de control para indicar que estamos mostrando el tercer dígito.
- `rcall sacanum` llama a `sacanum` para mostrar el valor en el display.

Resultado: El tercer dígito del `checksum_high` se muestra en el display.

---

5. Preparar y Mostrar el Cuarto Dígito del Checksum

```
assembly
Copiar código
mov r16, checksum_high
andi r16, 0b11110000
swap r16
ori r16, 0b10000000
rcall sacanum
```

Aquí se prepara el cuarto dígito:

- `mov r16, checksum_high` copia nuevamente el valor de `checksum_high` en `r16`.
- `andi r16, 0b11110000` aplica una **máscara** para extraer solo los 4 bits más significativos de `checksum_high`.
- `swap r16` intercambia los 4 bits altos y bajos de `r16`. Esto mueve los 4 bits altos a los 4 bits bajos, para que estén en la posición correcta para mostrarse en el display.
- `ori r16, 0b10000000` configura el bit de control para indicar que estamos mostrando el cuarto dígito.

- `rcall sacanum` llama a `sacanum` para mostrar el valor en el display.

Resultado: El cuarto dígito del `checksum_high` se muestra en el display.

---

6. Restaurar el Registro `r16` (Pop) y Retornar de la Interrupción

```
assembly
Copiar código
pop r16
reti
```

- `pop r16` restaura el valor original de `r16` desde la pila.
- `reti` (Return from Interrupt) finaliza la rutina de interrupción, permitiendo que el microcontrolador regrese al código que estaba ejecutando antes de que ocurriera la interrupción.

EXPLICACION PCINT1

```
_pcint1:
sbis PINC,  1 ;si el boton1 esta presionado
inc r26
sbis PINC, 2 ;sino, si boton2 esta presionado
inc r26
sbis PINC, 3 ;sino, si boton 3 esta presionado
inc r26
reti
```

1. Verificar el Botón en `PC1`

```
assembly
Copiar código
    sbis PINC, 1            ; Si el botón en PC1 está presi
onado
    inc r26
```

- `sbis PINC, 1` es una instrucción que significa "Skip if Bit in I/O Register is Set" (salta la siguiente instrucción si el bit especificado está en 1).
  - En este caso, verifica el bit 1 del registro `PINC`, que representa el pin `PC1`.
  - Los botones en este hardware son **activos en nivel bajo**, lo que significa que el pin `PC1` será `0` cuando el botón esté presionado y `1` cuando esté suelto.
  - **Si el botón no está presionado** (es decir, si `PC1` está en `1`), `sbis` salta la instrucción `inc r26` y pasa al siguiente botón.
  - **Si el botón está presionado** (es decir, si `PC1` está en `0`), `sbis` **no salta** y ejecuta la instrucción `inc r26`.
- `inc r26` incrementa el registro `r26` en `1`.
  - Esto establece el bit `0` de `r26` en `1` (si originalmente estaba en `0`), lo cual actúa como una **bandera** para indicar que un botón ha sido presionado.

2. Verificar el Botón en `PC2`

```
assembly
Copiar código
    sbis PINC, 2            ; Sino, si el botón en PC2 está
presionado
    inc r26
```

- `sbis PINC, 2` verifica el bit 2 del registro `PINC`, que representa el pin `PC2`.
  - **Si el botón en `PC2` no está presionado** (es decir, si `PC2` está en `1`), `sbis` salta la instrucción `inc r26` y continúa.
  - **Si el botón en `PC2` está presionado** (es decir, si `PC2` está en `0`), `sbis` no salta y se ejecuta `inc r26`.
- `inc r26` incrementa el registro `r26` en `1`.
  - Esto establece el bit `0` de `r26` en `1` (si originalmente estaba en `0`), indicando que un botón ha sido presionado.

3. Verificar el Botón en `PC3`

```
assembly
Copiar código
    sbis PINC, 3            ; Sino, si el botón en PC3 está
presionado
    inc r26
```

- `sbis PINC, 3` verifica el bit 3 del registro `PINC`, que representa el pin `PC3`.
  - **Si el botón en `PC3` no está presionado** (es decir, si `PC3` está en `1`), `sbis` salta la instrucción `inc r26` y sigue al final de la rutina.
  - **Si el botón en `PC3` está presionado** (es decir, si `PC3` está en `0`), `sbis` no salta y se ejecuta `inc r26`.
- `inc r26` incrementa el registro `r26` en `1`.
  - Esto también establece el bit `0` de `r26` en `1`, indicando que un botón ha sido presionado.

---

4. Finalizar la Interrupción

```
assembly
Copiar código
reti
```

- `reti` (Return from Interrupt) finaliza la rutina de interrupción y permite al microcontrolador regresar al código principal que estaba ejecutando antes de que ocurriera la interrupción.
- Al usar `reti`, el microcontrolador vuelve automáticamente al estado en el que estaba antes de la interrupción.

Resumen de la Rutina `_pcint1`

1. **Detecta si algún botón ha sido presionado** al verificar los pines `PC1`, `PC2`, y `PC3`.
   - Si el botón en `PC1` está presionado, incrementa `r26`.

- Si el botón en `PC2` está presionado, incrementa `r26`.
- Si el botón en `PC3` está presionado, incrementa `r26`.

2. **Usa `r26` como una bandera**:
   - Cada vez que un botón es presionado, el bit `0` de `r26` se pone en `1`.
   - Esta bandera puede ser utilizada en otras partes del código (por ejemplo, en `modo_receptor`) para detectar que un botón ha sido presionado.
3. **Termina la interrupción** con `reti` para que el microcontrolador pueda continuar su ejecución normal.

¿Cómo se usa `r26` en el código principal?

En el código de `modo_receptor`, `r26` se usa para detectar si algún botón ha sido presionado:

```
assembly
Copiar código
ldi r26, 0          ; Inicializa r26 en 0 al inicio
wait_4RX:
    sbrs r26, 0     ; Salta si el bit 0 de r26 es 1 (indic
a que se presionó un botón)
    rjmp wait_4RX   ; Si el bit 0 de r26 es 0, sigue esper
ando
```

- `sbrs r26, 0` verifica el bit `0` de `r26`.
  - Si `r26` tiene el bit `0` en `1` (es decir, algún botón fue presionado), `sbrs` salta la siguiente instrucción y el código puede continuar.
  - Si `r26` tiene el bit `0` en `0`, significa que ningún botón ha sido presionado, por lo que `rjmp wait_4RX` sigue en el bucle de espera.

EXPLICACION DEL CODIGO ESCLAVO:

```
; Laboratorio 5 , por ahora contiene las siguientes rutinas
;
```

```
; aleatorios - genera el vector de 512 números pseudoaleatori
;              XORSHIFT de 32 bits (https://en.wikipedia.org/
```

```
;
; Chksum_512 - Calcula el checksum de los 512 bytes en buffer
;              guarda el resultado en checksum_high:checksum_
;
```

```
; TX_512      - Transmite por el USART, los 1024 bytes de buf
;
; RX_512      - Recibe por el usart, los 1024 bytes.
;               que transmite la otra placa y lo coloca en buf
;
```

```
; _pcint1     - Rutina de atención a la interrupción de los b
;               si algún botón está apretado pone el bit0 de
```

```
;
; _tmr0_int   - Rutina de atención a la interupción del timer
;               segundo. Esta rutina saca por el disply check
```

```
;
;
; Registros reservados (uso global):
; checksum_high:checksum_low - Cheksum de 16 bits, es necesar
; r25             - Contiene el dígito en el display que esto
; r26             - Bit0: indica si se apretó un botón.
;
; Otros:
; r19:r18:r17:r16 - semilla de los números pseudoaleatorios .
;                   solo los usa aleatorios cuando está gener
```

```
; Definiciones de registros para facilitar su identificación
.def checksum_low = r4       ; r4 será llamado checksum_low
```

```
.def checksum_high = r5      ; r5 será llamado checksum_high
.def temp_reg = r1           ; r1 será llamado temp_reg
.def datoTransmitido = r0    ; Definición ya existente para r
```

```
; comienzo del código
.ORG 0x0000
jmp start         ; dirección de comienzo (vector de reset)
```

```
.ORG 0x0008
jmp _pcint1       ; salto a la rutina de atención a pcint1, i
.ORG 0x001C
jmp _tmr0_int     ; salto atención a rutina de comparación A
; ------------------------------------------------------------
```

```
; memoria RAM
.DSEG
buffer_msg: .byte 512         ; reservo 512 bytes para el vec
bmsg_end:   .byte 1           ; solo para marcar el final del
```

```
; comienzo del programa principal
.CSEG
start:
```

```
call system_init
ldi r26, 0x00             ; bandera teclado
mov checksum_high, r26    ; checksum a 0
mov checksum_low, r26
sei                       ; habilito interrupciones para disp
```

```
;jmp modo_transmisor
jmp modo_receptor
```

```
;modo_transmisor:
;   ldi r16, 0xA3             ; semilla de los números seudo-
```

```
;  ldi r17, 0x82
; ldi r18, 0xF0
;ldi r19, 0x05
;modo_transmisor_2:
;   rcall aleatorios          ; Genero los números aleatorios
```

; rcall Chksum_512 ; Genero Checksum

```
; ldi r26, 0
;wait_4TX:                    ; espero que alguien presione
;   sbrs r26, 0              ; Nota: la interrupcion del bot
```

; rjmp wait_4TX

```
; cli                        ; deshabilito interrupciones para
;rcall TX_512
;sei                        ; habilito interrupciones para dis
```

```
;rjmp modo_transmisor_2    ; empiezo todo de nuevo
```

;----------

modo_receptor:

```
ldi     r26,    0
```

```
wait_4RX:                          ;acá me pongo a esperar q
sbrs    r26,    0              ;Nota: la interrupcion del bo
rjmp    wait_4RX
```

```
;ahora recibo 512 bytes y los dejo en buffer_msg
lds     r16,    UDR0           ;me aseguro que el buffer est
lds     r16,    UDR0
lds     r16,    UDR0
cli                            ;deshabilito interrupciones p
rcall   RX_512                     ;recibo 512 bytes por pol
sei                                ;habilito interrupciones
```

```
rcall   Chksum_512                  ;calculo el nuevo Cheksum
rjmp    modo_receptor
```

```
;-------------------------------------------------------------
; Chksum - calcula el Checksum del vector buffer_msg (512 val
;-------------------------------------------------------------
Chksum_512:
ldi YL, low(buffer_msg)        ; Apunto Y al primer byte del
ldi YH, high(buffer_msg)
clr checksum_high              ; Inicializo el checksum
clr checksum_low
```

chksum_loop:
ld datoTransmitido, Y+ ; Traigo 1 byte a sumar
add checksum_low, datoTransmitido ; Sumo el byte al checksum (parte baja)
adc checksum_high, temp_reg ; Propago el acarreo a la parte alta del checksum
cpi YL, low(bmsg_end) ; Compruebo si llegué al final del buffer
brne chksum_loop
cpi YH, high(bmsg_end)
brne chksum_loop
ret

;--
; TX - rutina de transmisión serial USART. Transmite los 512 bytes de buffer_msg
;--
;TX_512:
; ldi ZL, low(buffer_msg) ; Apunto Z al primer byte del vector de 512 bytes
; ldi ZH, high(buffer_msg)
; ldi r16, (1 << TXEN0) ; Habilito el transmisor USART
;sts UCSR0B, r16

;TX_loop1:
; ld datoTransmitido, Z+ ; Traigo el Byte a transmitir
; sts UDR0, datoTransmitido ; Pongo a transmitir (UDR0)

;TX_loop2:
; lds r16, UCSR0A ; Espero a que termine la transmisión del byte por polling (UCSR0A)
; sbrs r16, UDRE0
; rjmp TX_loop2

```
;cpi ZL, low(bmsg_end)          ; Chequeo si llegué al fina
l del buffer
;brne TX_loop1
;cpi ZH, high(bmsg_end)
;brne TX_loop1

;ret
```

;chequeo si llegué al final del buffer

```
;ret
```

;---
; RX - rutina de recepción usart. Recibe 1024 bytes y los deja en de buffer_msg
;---
RX_512:
ldi ZL, low(buffer_msg) ; Apunto Z al primer byte del vector de 512 bytes
ldi ZH, high(buffer_msg)
ldi r16, (1 << RXEN0) ; Configuro el USART como receptor
sts UCSR0B, r16

RX_Wait:
lds r16, UCSR0A ; Ahora polling para esperar recibir algo (UDR0)
sbrs r16, RXC0
rjmp RX_Wait

```
lds     datoTransmitido, UDR0                  ; Llego aqu
í solo si recibí algo
st      Z+, datoTransmitido                    ; Guardo lo
que recibí

cpi     ZL, low(bmsg_end)          ; Chequeo si llegué al f
inal del buffer
```

```
 brne    RX_Wait
 cpi     ZH, high(bmsg_end)
 brne    RX_Wait

 ret
```

//--
system_init:
;configuro los puertos:
;	PB2 PB3 PB4 PB5 - son los LEDs del shield
;	PB0 es SD (serial data) para el display 7seg
;	PD7 es SCLK, el reloj de los shift registers del display 7seg
;	PD4 transfiere los datos que ya ingresaron en serie, a la salida del registro paralelo

```
 ldi     r16,    0b00111101
 out     DDRB,   r16         ;4 LEDs del shield son salidas
 out     PORTB,  r16         ;apago los LEDs
 ldi     r16,    0b00000000
 out     DDRC,   r16         ;3 botones del shield son entra
 das
 ldi     r16,    0b10010001
 out     DDRD,   r16         ;configuro PD.0, PD.4 y PD.7 co
 mo salidas
 cbi     PORTD,  7           ;PD.7 a 0, es el reloj serial d
 el  Display, inicializo a 0
 cbi     PORTD,  4           ;PD.4 a 0, es el reloj del latc
 h del Display, inicializo a 0
```

;---

;Configuro interrupcion por cambio en PC.1, PC.2, PC.3.
ldi		r16, 0b00000010
sts		PCICR,	r16		;habilito PCI1 que es (PCI8:PCI14)
ldi		r16, 0b00001110
sts		PCMSK1, r16		;habilito detectar cambios en PortC 1,2,3 (PCI9,PCI10,PCI11)
;---

```
----
;Configuro el TMR0 y su interrupcion.
ldi         r16,  0b00000010
out         TCCR0A, r16             ;configuro para que cuente hasta OCR0A y
vuelve a cero (reset on compare), ahí dispara la interrupción
ldi         r16,  0b00000010
out         TCCR0B, r16             ;prescaler = 256
ldi         r16,  249
out         OCR0A,   r16            ;comparo con 249
ldi         r16,  0b00000010
sts         TIMSK0,  r16            ;habilito la interrupción (falta habilitar global)
;-------------------------------------------------------------------------------
----
;Inicializo USART para transmitir
ldi         r16,  0b00010000
sts         UCSR0B, r16
ldi         r16,  0b00000110
sts         UCSR0C,       r16
ldi         r16,  0x00              //9600 baudios (baudio = bit/segundo)
sts         UBRR0H, r16
ldi         r16,  0x67
sts         UBRR0L, r16

;-------------------------------------------------------------------------------
----
;Inicializo algunos registros que voy a usar como variables.
ldi         r25, 0x10               ;inicializo r25 para el display r25 = 00010000 ;
00100000 ; 01000000 ; 10000000 indica qué digito sale
;-------------------------------------------------------------------------------
----
;Fin de la inicialización
ret

;-------------------------------------------------------------------------------
----
;                           *               RUTINAS                 *
;-------------------------------------------------------------------------------
----
```

```
;-----------------------------------------------------------
; ALEATORIOS
;-----------------------------------------------------------
;rutina que genera 512 bytes pseudoaleatorios en buffer_msg

aleatorios:
ldi         r28, low(buffer_msg)          ;apunto Y al primer byte del mensaje
ldi         r29, high(buffer_msg)
ale_loop:
; genero un número de 32bits nuevo usando XORSHIFT de 32 bits (
https://en.wikipedia.org/wiki/Xorshift)
ldi         r20, 13
call  ale_loop_l
ldi         r20, 17
call  ale_loop_r
ldi         r20, 5
call  ale_loop_l
;------ acá ya tengo el numero pseudo-aleatorio de 32bits, voy a guardar los
32 bits (podria solo ir guardando de a 8)
```

```
st      Y+,     r16                 ;el número aleatorio lo
guardo a partir de adonde apunta el registro Y, voy recorri
endo hasta 512
st      Y+,     r17
st      Y+,     r18
st      Y+,     r19
cpi     YL,     low(bmsg_end)
brne    ale_loop
cpi     YH,     high(bmsg_end)
brne    ale_loop
ret
```

```
ale_loop_l:
mov       datoTransmitido,        r16
mov       r1,         r17
mov       r2,         r18
mov       r3,         r19
ale_rota_l:
clc
```

```
rol         datoTransmitido
rol         r1
rol         r2
rol         r3
dec         r20
brne        ale_rota_l
rjmp        ale_rota_out

ale_loop_r:
mov         datoTransmitido,        r16
mov         r1,         r17
mov         r2,         r18
mov         r3,         r19
ale_rota_r:
clc
ror         r3
ror         r2
ror         r1
ror         datoTransmitido
dec         r20
brne        ale_rota_r

ale_rota_out:
eor         r16,        datoTransmitido
eor         r17,        r1
eor         r18,        r2
eor         r19,        r3
ret

;-----------------------------------------------------------------------------
----
;   SACANUM
;-----------------------------------------------------------------------------
----
; rutina que saca un número por el display

sacanum:
push r16            ; guardo una copia de r16
ldi zh, high(segmap<<1) ; Initialize Z-pointer
ldi zl, low(segmap<<1)
andi r16, 0x0F
```

```
add zl, r16
clr r16
adc zh, r16
lpm r16, Z          ; traigo de la memoria de Programa el 7-Seg
call sacabyte
pop r16
call sacabyte
sbi PORTD, 4        ; PD.4 a 1, es el reloj del latch
cbi PORTD, 4        ; PD.4 a 0, es el reloj del latch
ret
```

```
;-----------------------------------------------------------
;   SACABYTE
;-----------------------------------------------------------
; saca un byte por el 7seg
sacabyte:
ldi r17, 0x08
loop_byte1:
cbi PORTD, 7            ; SCLK = 0
lsr r16
brcs loop_byte2         ; salta si C=1
cbi PORTB, 0            ; SD = 0
rjmp loop_byte3
loop_byte2:
sbi PORTB, 0            ; SD = 1
loop_byte3:
sbi PORTD, 7            ; SCLK = 1
dec r17
brne loop_byte1
ret
```

segmap:

```
.db 0b00000011, 0b10011111, 0b00100101, 0b00001101 ;"0" "1" "
.db 0b10011001, 0b01001001, 0b01000001, 0b00011111 ;"4" "5" "
.db 0b00000001, 0b00001001, 0b00010001, 0b11000001 ;"8" "9" "
.db 0b01100011, 0b10000101, 0b01100001, 0b01110001 ;"C" "d" "
```

; Rutina de atención a la interrupción del Timer0.
_tmr0_int:

```
push r16
ldi r16,0
mov r16, checksum_low
andi r16, 0x0F
ori r16,0b00010000
rcall sacanum
mov r16,checksum_high
andi r16, 0x0F
swap r16
ori r16,0b00100000
rcall sacanum
mov r16, checksum_high
andi r16, 0x0F
ori r16,0b01000000
rcall sacanum
```

```
mov r16,checksum_high
andi r16, 0b11110000
swap r16
ori r16,0b10000000
rcall sacanum
```

```
pop r16
reti
```

```
; ----------------------------------------------------------
; Rutina de atención a la interrupción por cambio en el estad
; ----------------------------------------------------------
; recordar que se configuró la detección por cambio para que
; se dispara la interrupción. LA interrupción no distingué qu
; Los botones se encuentran en PC.1, PC.2, PC.3 y recordar de
;
_pcint1:
```

```
sbis PINC,  1 ;si el boton1 esta presionado
inc r26
sbis PINC, 2 ;sino, si boton2 esta presionado
inc r26
sbis PINC, 3 ;sino, si boton 3 esta presionado
inc r26
reti
```

1. Vectores de Interrupción y Definiciones de Memoria

```
.ORG 0x0000
    jmp start          ; dirección de comienzo (vector de re
set)
.ORG 0x0008
    jmp _pcint1        ; salto a la rutina de atención a pci
nt1, interrupción por cambio para los botones
.ORG 0x001C
    jmp _tmr0_int      ; salto atención a rutina de comparac
ión A del timer 0
```

- `.ORG` define las **direcciones de memoria** donde comienzan las interrupciones.
- `jmp start`: Este es el **vector de reset**. Cuando el microcontrolador se reinicia, se ejecuta el código en `start`.
- `jmp _pcint1`: Esta es la **interrupción por cambio de pin** en `PC1`, `PC2`, y `PC3`, que activará la rutina `_pcint1`.
- `jmp _tmr0_int`: Es la **interrupción del temporizador 0 (Timer0)**, que llamará a la rutina `_tmr0_int` cada cierto intervalo de tiempo.

2. Segmento de Datos en la Memoria RAM

```
.DSEG
buffer_msg: .byte 512          ; reservo 512 bytes para el v
ector de números aleatorios a transmitir.
```

```
bmsg_end:   .byte 1           ; solo para marcar el final d
el buffer
```

- `.DSEG` define el **segmento de datos** en RAM.
- `buffer_msg` reserva 512 bytes en RAM para almacenar números aleatorios.
- `bmsg_end` es una marca de un byte después del buffer para indicar el final de `buffer_msg`.

---

3. Código Principal (`start`)

```
.CSEG
start:
    call system_init
    ldi r26, 0x00             ; bandera teclado
    mov checksum_high, r26    ; checksum a 0
    mov checksum_low, r26
    sei                       ; habilito interrupciones par
a display y botones
    jmp modo_receptor
```

- `.CSEG` define el **segmento de código** en memoria de programa.
- `start` es el punto de entrada principal del programa.
- `call system_init`: Llama a `system_init`, que configura los puertos, interrupciones y el temporizador.
- `ldi r26, 0x00`: Inicializa `r26` en 0, que se usará como **bandera de estado** para los botones.
- `mov checksum_high, r26` y `mov checksum_low, r26`: Inicializan el checksum en cero.
- `sei` (Set Interrupts): Habilita las interrupciones globalmente.
- `jmp modo_receptor`: Salta a `modo_receptor` para iniciar en modo receptor.

---

4. `modo_receptor`

```
modo_receptor:
    ldi r26, 0              ; Reinicia el valor de `r26`.
```

```
wait_4RX:
    sbrs r26, 0             ; Salta si el bit 0 de `r26` e
s 1 (indica que se presionó un botón).
    rjmp wait_4RX           ; Si el bit 0 de `r26` es 0, s
igue esperando en un bucle.

    ; Preparación para recibir datos
    lds r16, UDR0           ; Lee tres veces de `UDR0` par
a vaciar el buffer de recepción.
    lds r16, UDR0
    lds r16, UDR0
    cli                     ; Deshabilita las interrupcion
es temporalmente.
    rcall RX_512            ; Llama a la subrutina `RX_512
` para recibir 512 bytes de datos.
    sei                     ; Habilita las interrupciones
nuevamente.
    rcall Chksum_512        ; Llama a `Chksum_512` para ca
lcular el checksum de los datos recibidos.
    rjmp modo_receptor      ; Regresa al inicio de `modo_r
eceptor`.
```

- `ldi r26, 0`: Reinicia `r26` para asegurarse de que el bit 0 esté en 0 (sin presionar botón).
- `sbrs r26, 0` y `rjmp wait_4RX`: Espera hasta que se presione un botón (bit 0 de `r26` se pone en 1 por `_pcint1`).
- `lds r16, UDR0` (3 veces): Vacía el buffer de recepción USART leyendo `UDR0` tres veces.
- `cli` y `sei`: Deshabilita y luego habilita las interrupciones para proteger la recepción de datos.
- `rcall RX_512`: Llama a `RX_512` para recibir 512 bytes desde USART.
- `rcall Chksum_512`: Calcula el checksum de los datos recibidos y lo guarda en `checksum_high:checksum_low`.
- `rjmp modo_receptor`: Regresa al inicio de `modo_receptor`.

---

5. `Chksum_512`: Cálculo del Checksum

```
Chksum_512:
    ldi YL, low(buffer_msg)        ; Apunta Y al inicio de
`buffer_msg`.
    ldi YH, high(buffer_msg)
    clr checksum_high              ; Inicializa el checksum
en 0.
    clr checksum_low

chksum_loop:
    ld datoTransmitido, Y+         ; Carga un byte del buff
er a `datoTransmitido`.
    add checksum_low, datoTransmitido   ; Suma `datoTransmi
tido` a `checksum_low`.
    adc checksum_high, temp_reg    ; Propaga el acarreo a `
checksum_high`.
    cpi YL, low(bmsg_end)          ; Compara `YL` con el fi
nal del buffer.
    brne chksum_loop               ; Si no es el final, rep
ite el bucle.
    cpi YH, high(bmsg_end)
    brne chksum_loop
    ret
```

- Inicializa `Y` apuntando a `buffer_msg`, y `checksum_high` y `checksum_low` en cero.
- Bucle `chksum_loop`:
  - `ld datoTransmitido, Y+`: Carga un byte del buffer.
  - `add` y `adc`: Suma el byte al checksum y propaga el carry.
  - `cpi` y `brne`: Verifica si llegó al final del buffer; si no, repite.
- `ret`: Finaliza la subrutina y regresa.

---

6. `RX_512`: Recepción de Datos desde USART

```
RX_512:
    ldi ZL, low(buffer_msg)        ; Apunta Z al inicio de
`buffer_msg`.
```

```
    ldi ZH, high(buffer_msg)
    ldi r16, (1 << RXEN0)          ; Habilita el receptor U
SART.
    sts UCSR0B, r16

RX_Wait:
    lds r16, UCSR0A                ; Polling para verificar
si hay un byte disponible.
    sbrs r16, RXC0
    rjmp RX_Wait

    lds datoTransmitido, UDR0      ; Lee el byte recibido.
    st Z+, datoTransmitido         ; Guarda el byte en `buf
fer_msg`.

    cpi ZL, low(bmsg_end)          ; Verifica si llegó al f
inal del buffer.
    brne RX_Wait
    cpi ZH, high(bmsg_end)
    brne RX_Wait

    ret
```

- Inicializa `Z` apuntando a `buffer_msg`, y habilita el receptor USART.
- `RX_Wait`: Espera hasta que un byte esté disponible (polling).
- Lee el byte de `UDR0` y lo guarda en `buffer_msg`.
- Verifica si `Z` alcanzó `bmsg_end`; si no, sigue recibiendo.
- `ret`: Finaliza la subrutina.

---

7. `system_init`: Inicialización de Puertos y Configuración

```
system_init:
    ldi r16, 0b00111101
    out DDRB, r16              ; Configura PB2-PB5 como sali
das.
    out PORTB, r16             ; Inicializa los LEDs en apag
```

```
ado.
    ldi r16, 0b00000000
    out DDRC, r16            ; Configura botones en PC1-PC
3 como entradas.
    ldi r16, 0b10010001
    out DDRD, r16            ; Configura puertos del displ
ay.
    cbi PORTD, 7             ; Inicializa señal de reloj d
el display.
    cbi PORTD, 4             ; Inicializa latch del displa
y.
```

- Configura los **puertos**:
  - LEDs en `PB2-PB5` como salidas.
  - Botones en `PC1-PC3` como entradas.
  - Pines `PD0`, `PD4`, `PD7` para el display.
- Inicializa las señales del **display** (reloj y latch) en 0.

---

8. Configuración de Interrupciones y Temporizador

```
    ldi

 r16, 0b00000010
    sts PCICR, r16           ; Habilita interrupción en `P
CI1`.
    ldi r16, 0b00001110
    sts PCMSK1, r16          ; Activa interrupción por cam
bio en `PC1`, `PC2`, `PC3`.

    ldi r16, 0b00000010
    out TCCR0A, r16          ; Configura Timer0 para compa
rar con OCR0A.
    ldi r16, 0b00000010
    out TCCR0B, r16          ; Prescaler del Timer0 = 256.
    ldi r16, 249
    out OCR0A, r16           ; Configura el valor de compa
```

```
ración.
    ldi r16, 0b00000010
    sts TIMSK0, r16           ; Habilita la interrupción de
l Timer0.
```

- Configura **interrupción por cambio** para los botones en `PC1`, `PC2`, y `PC3`.
- Configura el **Timer0**:
  - Modo de comparación con `OCR0A`.
  - Prescaler de 256 y valor de comparación de 249 para definir la frecuencia de interrupción.

---

9. Configuración del USART para Transmisión

```
    ldi r16, 0b00010000
    sts UCSR0B, r16           ; Habilita transmisión USART.
    ldi r16, 0b00000110
    sts UCSR0C, r16           ; Configura 8 bits de datos.
    ldi r16, 0x00
    sts UBRR0H, r16           ; Baud rate alto.
    ldi r16, 0x67
    sts UBRR0L, r16           ; Baud rate bajo (9600 baudio
s).
```

- Configura el **USART** para transmisión a 9600 baudios.
