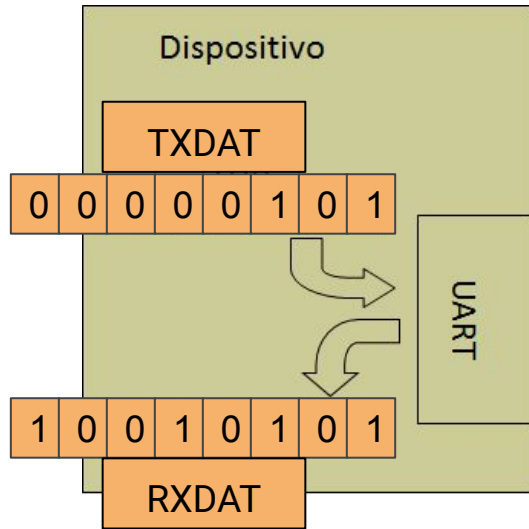


# Comunicación serie - primitivas y drivers

Informática II - R2004  
2021

# Universal Synchronic/Asinchronic Receiver Transmitter (USART)

Este dispositivo se encarga de la parte física de la comunicación, y nuestra tarea será configurarlo para que se comporte de la manera deseada en nuestro sistema.



# Registros configuración USART

Name	Access	Offset	Description
CFG	R/W	0x000	USART Configuration register. Basic USART configuration settings that typically are not changed during operation.
CTL	R/W	0x004	USART Control register. USART control settings that are more likely to change during operation.
STAT	R/W	0x008	USART Status register. The complete status value can be read here. Writing ones clears some bits in the register. Some bits can be cleared by writing a 1 to them.
INTENSET	R/W	0x00C	Interrupt Enable read and Set register. Contains an individual interrupt enable bit for each potential USART interrupt. A complete value may be read from this register. Writing a 1 to any implemented bit position causes that bit to be set.
INTENCLR	W	0x010	Interrupt Enable Clear register. Allows clearing any combination of bits in the INTENSET register. Writing a 1 to any implemented bit position causes the corresponding bit to be cleared.
RXDAT	R	0x014	Receiver Data register. Contains the last character received.
RXDATSTAT	R	0x018	Receiver Data with Status register. Combines the last character received with the current USART receive status. Allows DMA or software to recover incoming data and status together.
TXDAT	R/W	0x01C	Transmit Data register. Data to be transmitted is written here.

**CFG:** Configuración general de la comunicación

**STAT:** Estado de la comunicación (recepción, transmisión, errores)

**INTENSET:** Habilitación de las interrupciones (por tx, rx, error)

**INTENCLR:** Deshabilitación de las interrupciones (por tx, rx, error)

**RXDAT:** Buffer donde se recibe el dato que llega por puerto serie

**TXDAT:** Buffer donde se guarda el dato que se enviará por puerto serie

# Iniciando la UART

```
void UART0_Init(void)
```

```
{  
    // HABILITO LA UART0  
    SYSCON->SYSAHBCLKCTRL0 |= (1 << 14);
```

```
    //Reseteo la UART0:  
    SYSCON->PRESETCTRL0 &= ~(1<<14);  
    SYSCON->PRESETCTRL0 |= (1<<14);
```

```
    //Habilito el CLK a la SWM:  
    SYSCON->SYSAHBCLKCTRL0 |= (1<<7);
```

```
    // CONFIGURO LA MATRIX TX0 en P0.25 y RX0 en P0.24  
    PINASSIGN0 &= ~(0x0000FFFF);  
    PINASSIGN0 = (25 << 0) | (24 << 8);
```

```
    // CONFIGURACION GENERAL  
    USART0->CFG = (0 << 0)          // 0=DISABLE 1=ENABLE  
    | (1 << 2)          // 0=7BITS 1=8BITS 2=9BITS  
    | (0 << 4)          // 0=NOPARITY 2=PAR 3=IMPAR  
    | (0 << 6)          // 0=1BITSTOP 1=2BITSTOP  
    | (0 << 9)          // 0=NOFLOWCONTROL 1=FLOWCONTROL  
    | (0 << 11);        // 0=ASINCRONICA 1=SINCRONICA
```

```
    // CONFIGURACION INTERRUPTIONES  
    USART0->INTENSET = (1 << 0);    //RX
```

```
    // CONFIGURACION DEL BAUDRATE  
    UART0CLKSEL = 0; //CLK = FRO = 30MHz  
    USART0->BRG = (FREQ_PRINCIPAL / (BAUDRATE* 16));
```

```
    //Habilito interrupcion USART0 en el NVIC  
    ISER0 |= (1 << 3);
```

```
    // ENABLE LA UART  
    USART0->CFG |= 1;
```

```
    return;
```

**Una vez finalizada la configuración, habilito la UART**

## **SYSAHBCLKCTRL**

Habilito el CLK del periférico

## **PINASSIGN0**

Configuro el pin P0.8 para la transmisión y el P0.9 para la recepción del puerto serie

## **CFG - Configuración de la comunicación**

Selecciono una palabra de 8 bits, 1 bit de stop, sin control de paridad, configuro que la comunicación sea solo con 2 pines, en formato asincrónico.

## **INTENSET - Configuración de interrupciones**

Configuro el periférico para que interrumpa cada vez que llega un nuevo byte por el puerto serie

## **Baudrate - Velocidad de la comunicación**

Configuro el clk del periférico como el clk principal, y el divisor de la velocidad para alcanzar la velocidad deseada

## **NVIC - Habilidad de interrupción**

Configuro el controlador programable de interrupciones para que la interrupción de la USART0 interrumpa el programa

# USARTn\_IRQHandler - Driver

```
void USART0_IRQHandler (void)
```

```
{
```

```
    int16_t auxTemporal;
```

```
    uint32_t stat = USART0->STAT;
```

```
    // CASO RECEPCION
```

```
    if(stat & (1 << 0))
```

```
    {
```

```
    }
```

```
    //CASO TRANSMISION
```

```
    if(stat & (1 << 2))
```

```
    {
```

```
    }
```

```
    //CASO ERRORES
```

```
    else
```

```
    {
```

```
    }
```

```
}
```

Una vez que se genera una interrupción, debo saber si la misma se generó por recepción (llegó un nuevo dato), por transmisión (se terminó de enviar un dato), o por error (en caso de que haya activado este tipo de interrupción). Esto puedo verlo en el registro **STAT**

b16	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
ABERR	NOISE	PARITY	FRAME	START	DELTA	RXBR FAK	--	OVERRU NINT	--	TXDISS TAT	DELTA CTS	CTS	TXIDLE	TXRDY	RXIDLE	RXRDY

# ¿Cómo hacemos una estrategia para mandar muchos datos por el puerto?

APLICACIÓN



1. La aplicación llama a una función que “envía” datos por el puerto:

```
Transmitir ( uint8_t * texto);
```

```
Por ej: Transmitir (“Hola Mundo”);
```

PRIMITIVAS

DRIVERS

HARDWARE



# ¿Cómo hacemos una estrategia para mandar muchos datos por el puerto?

APLICACIÓN

PRIMITIVAS

DRIVERS

HARDWARE



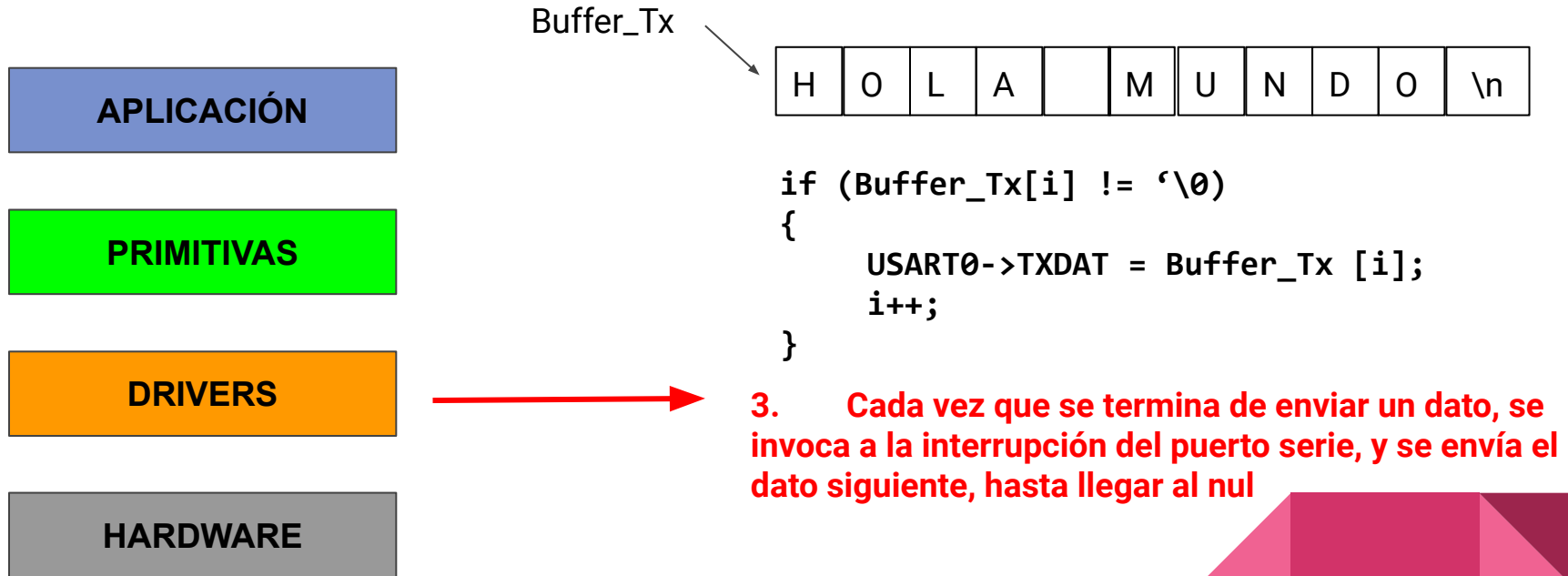
Buffer\_Tx

2. La función Primitiva Transmitir guarda todos los datos a enviar en un buffer, y habilita la interrupción por transmisión (para comenzar el envío):

H	O	L	A		M	U	N	D	O	\n
---	---	---	---	--	---	---	---	---	---	----

USART0->INTENSET |= 1<<2;

# ¿Cómo hacemos una estrategia para mandar muchos datos por el puerto?





# ¿Qué problemas tiene la estrategia anterior?

Para poder llamar a Transmitir, tengo que saber que no hay datos esperando a ser transmitidos (sino se pisan)

Por ejemplo, si invoco a transmitir de esta manera:


```
Transmitir (“Hola”);  
Transmitir (“Mundo”);
```

Se enviaría la H de Hola, y a continuación la palabra Mundo.  
Además, no puedo enviar un dato a menos que sepa que el buffer de transmisión no está enviando nada en ese momento (THRE)

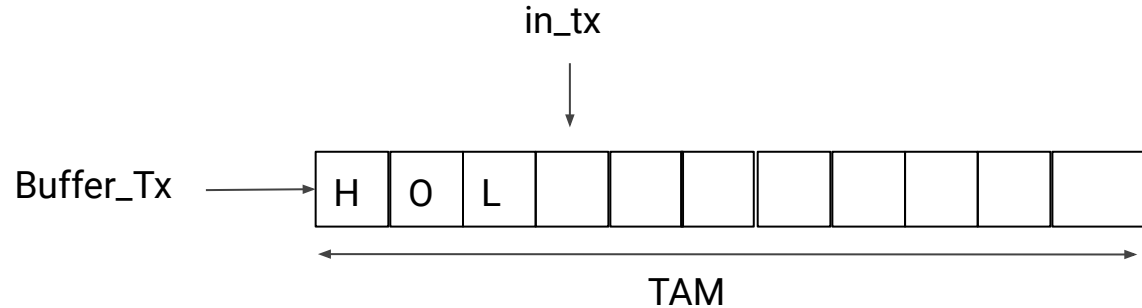
# Solución: Buffers circulares

Utilizaré un buffer que vaya almacenando (***push***) los datos que se quieren enviar en forma global. La interrupción del puerto serie irá sacando los datos de este buffer (***pop***) de acuerdo a la velocidad a la que se vayan enviando. Cuando se llegue al final del buffer (la última posición del vector), se volverá a empezar a llenarlo desde el comienzo, entendiendo que los primeros datos ya fueron enviados:

```
Transmitir ( uint8_t * datos, uint8_t cantidad)
{
    uint8_t i;
    for (i = 0 ; i < cantidad ; i++)
        Push(datos[i]);
}
```

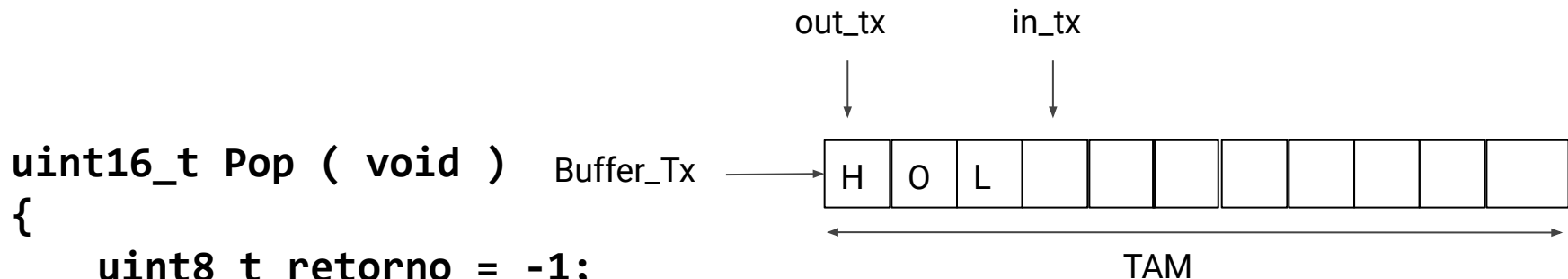


# Función Push - poner un dato en el buffer circular



```
Push ( uint8_t dato )  
{  
    static uint8_t in_tx = 0;  
    Buffer_Tx[in_tx] = dato;  
    in_tx ++;  
    if ( in_tx == TAM )  
        in_tx = 0;  
}
```

# Función Pop - Sacar un dato del buffer circular



```
uint16_t Pop ( void )  
{  
    uint8_t retorno = -1;  
    if ( out_tx != in_tx )  
    {  
        retorno = Buffer_tx[out_tx];  
        out_tx++;  
        out_tx %= TAM;  
    }  
    return retorno;  
}
```

**¿Cómo se si el buffer está  
lleno o vacío?  
COMO CONDICIÓN DE DISEÑO EL  
BUFFER NO DEBERÍA PODER  
LLENARSE**

# Ventajas y desventajas de la estrategia de buffers circulares

Utilizando buffers circulares no tengo que esperar a que un dispositivo de menor velocidad (la USART) termine una operación para volver a enviar un dato

Sin embargo, no puedo enviar demasiados datos a mucha velocidad con la función Transmitir, porque sino el buffer se llena.

En general, los buffers circulares se utilizan para adaptar dos dispositivos que funcionan a velocidades distintas (el buffer “le da tiempo” al dispositivo más lento a que se envíen los datos mientras el dispositivo más rápido sigue haciendo otra cosa.

# ¿Cómo quedarían las funciones (primitivas) para envío de datos?

```
main():  
while (1)  
{  
    ...  
    if ( GetKey() == SW0 )  
        Transmitir("Hola Mundo",10);  
    ...  
}
```

```
Transmitir ( uint8_t * datos, uint8_t cantidad)  
{  
    uint8_t i;  
    for (i = 0 ; i < cantidad ; i++)  
        PushTx(datos[i]);  
}
```

```
void PushTx ( uint8_t dato )  
{  
    Buffer_Tx[in_tx] = dato;  
    in_tx++;  
    in_tx %= TAM;  
  
    //Para que el dato se mande, activo la interrupción por tx:  
    USART0->INTENSET |= 1<<2;  
}
```

# ¿Cómo quedarían las funciones (drivers) para envío de datos?

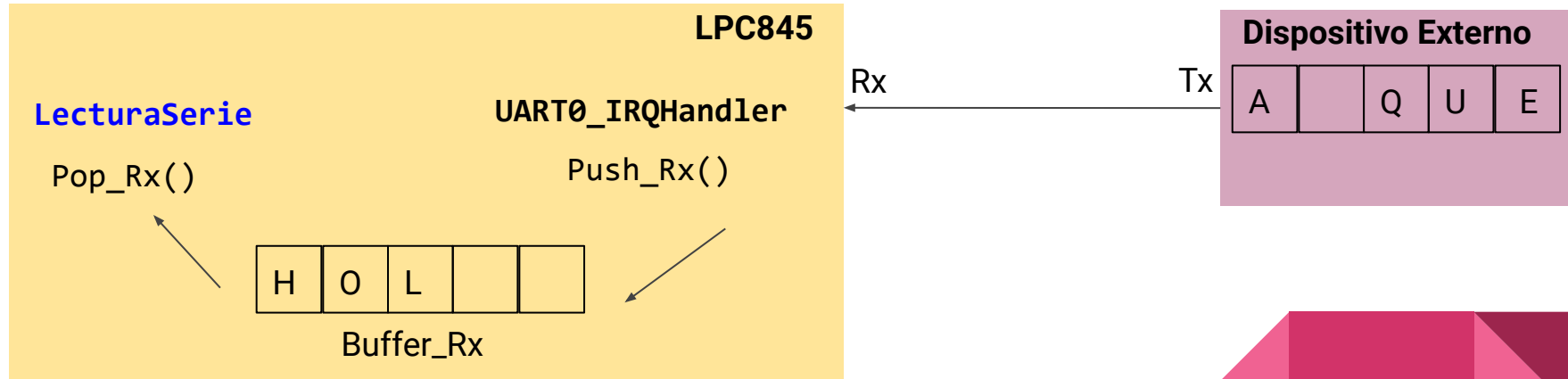
```
void UART0_IRQHandler (void)
{
    int16_t estado, dato;
    estado = USART0->STAT;

    if( estado & 0x04 ) //TX
    {
        dato = PopTx();
        if ( dato != -1 )
            USART0->TXDAT = dato;
        else
            USART0->INTENCLR = 1<<2;
    }
    ...
}
```

```
int16_t PopTx ( void )
{
    int16_t retorno = -1;
    if ( out_tx != in_tx )
    {
        retorno = Buffer_tx[out_tx];
        out_tx++;
        out_tx %= TAM;
    }
    return retorno;
}
```

# Buffers circulares para la recepción de datos

De la misma manera que hay buffers para el envío de datos, pueden usarse para la recepción también. A medida que va llegando un dato por la interrupción de la USART, se va cargando un buffer con la función `Push_Rx()`, que el programa irá analizando en una máquina de estados dedicada utilizando la función `Pop_Rx()`



La UART interrumpe por cada byte que llega. El tiempo entre cada interrupción va a depender de la velocidad de transmisión y cuando el dispositivo externo decida mandar cada dato.



# Estrategia para recibir datos por el puerto serie

APLICACIÓN

PRIMITIVAS

DRIVERS

HARDWARE

```
void UART0_IRQHandler (void)
{
    int16_t estado, dato;
    estado = USART0->STAT;
    ...

    if( estado & 0x01 ) //RX
    {
        Push_Rx(USART0->RXDAT);
    }
}
```

```
void Push_Rx ( uint8_t dato )
{
    Buffer_Rx[in_rx] = dato;
    in_rx++;
    in_rx %= TAM;
}
```



1. Se recibe la interrupción de la UART para recepción. Guardo el dato que recibo en el buffer circular *Buffer\_Rx*.

# Buffers circulares para la recepción de datos

APLICACIÓN

PRIMITIVAS

DRIVERS

HARDWARE



2. La función Primitiva lee el buffer circular. Si hay datos devuelve el que corresponde de out\_rx, si no hay dato devuelve -1.

```
int16_t PopRx ( void )
{
    int16_t retorno = -1;
    if ( out_rx != in_rx )
    {
        retorno = Buffer_rx[out_rx];
        out_rx++;
        out_rx %= TAM;
    }
    return retorno;
}
```

# Buffers circulares para la recepción de datos

APLICACIÓN



3. La aplicación utiliza la función **Pop\_Rx** para ir leyendo los datos. Es muy importante chequear que el dato devuelto sea válido (o sea != de -1)

PRIMITIVAS

DRIVERS

HARDWARE

```
void main (void)
{
    Inicializacion();
    while ( 1 ) {
        ...

        LecturaSerie();
        ...
    }
}
```

```
void LecturaSerie()
{
    dato = Pop_Rx();
    if ( dato != -1 )
    {
        //Llego un dato nuevo
    }
}
```

# Transmisión y Recepción conjunta

```
void USART0_IRQHandler (void)
{
    uint16_t estado, dato;
    estado = USART0->STAT;

    if( estado & 0x04 ) //TX
    {
        dato = PopTx();
        if ( dato != -1 )
            USART0->TXDAT = dato;
        else
            primerDato = 1;
    }

    if( estado & 0x01 ) //RX
    {
        Push_Rx(USART0->RXDAT);
    }
}
```

**Si quisiera habilitar la interrupción por errores debería hacer un if por cada tipo de error habilitado, y hacer el tratamiento que considere correspondiente a dicho error**

b16	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
ABERR	NOISE	PARITY	FRAME	START	DELTA	RXBR EAK	--	OVERRU NINT	--	TXDISS TAT	DELTA CTS	CTS	TXIDLE	TXRDY	RXIDLE	RXRDY

# Errores en las transmisiones

Al estar comunicando dos dispositivos que están separados físicamente, crecen mucho las posibilidades de que haya errores en la recepción de los datos. Para verificar que los mismos sean correctos, muchas veces se arman “**tramas**” de datos, en los que un dato tiene un valor fijo que indica que la trama está por empezar o que ya terminó, o se puede enviar un dato cuyo valor dependa de los valores anteriores (la suma, por ejemplo) para verificar que todos llegaron bien.

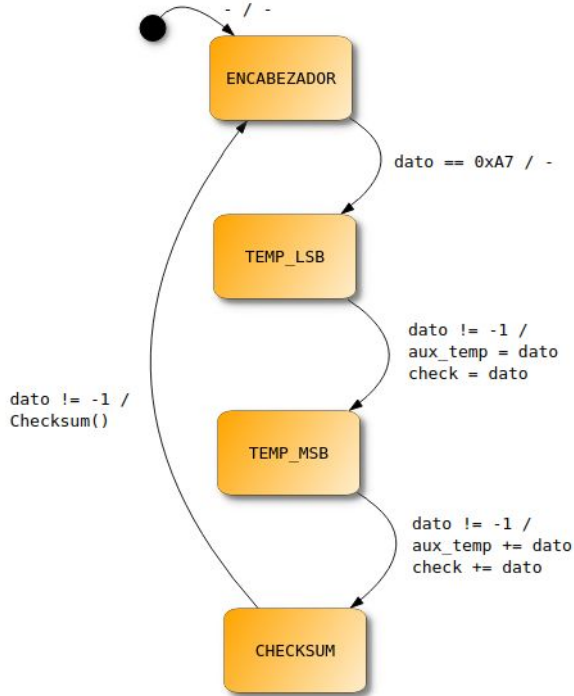
ENCABEZADOR (0xA7)	Temperatura (8 bits - LSB)	Temperatura (8 bits - MSB)	CHECKSUM (suma de los anteriores)
-----------------------	-------------------------------	-------------------------------	--------------------------------------



# Función Aplicación Recepción (ejemplo)

Trama Recibida:

ENCABEZADOR (0xA7)	Temperatura (8 bits - LSB)	Temperatura (8 bits - MSB)	CHECKSUM (suma de los anteriores)
-----------------------	-------------------------------	-------------------------------	--------------------------------------



```
void LecturaSerie ( void )
{
    static uint8_t Estado;
    if ( ( dato = Pop_Rx() ) != -1 )
    {
        switch ( Estado )
        {
            case ENCABEZADOR :
                if ( dato == 0xA7 )
                    Estado = TEMP_LSB;
                break;

            case TEMP_LSB :
                aux_temp = dato;
                check = dato;
                Estado = TEMP_MSB;
                break;

            case TEMP_MSB :
                aux_temp |= dato << 8;
                check += dato;
                Estado = CHECKSUM;
                break;

            case CHECKSUM:
                Checksum(check, dato, aux_t);
                Estado = ENCABEZADOR;
                break;
        }
    }
}
```

```
void Checksum(uint8_t check, uint8_t dt, uint8_t aux_t)
{
    if( check == dt )
    {
        temperatura = aux_t;
    }
}
```

La variable temperatura es una variable global donde se guarda el dato una vez verificada TODA la trama.