

Sistemas Empotrados

Tema 5: Entrada/salida

Lección 13:
Diseño de drivers de nivel 0 y 1



Contenidos

Tema 5: Entrada/salida

Introducción

El GPIO

Acceso al mapa de memoria

Diseño de drivers en capas

Drivers de nivel 0

Introducción

Diseño de un driver L0

Drivers de nivel 1

Introducción

Diseño de un driver L1

Drivers de nivel 2

Introducción

Diseño de un driver L2

Drivers de nivel 0



Drivers de nivel bajo, que permiten la implementación de un sistema sencillo

Características:

- Tamaño pequeño del código
- Con muy poco o ningún chequeo de errores
- Sólo soportan las características principales del dispositivo
- Sin soporte de parámetros de configuración del dispositivo
- Sólo soportan E/S controlada por programa
- Las llamadas a las funciones del driver son bloqueantes

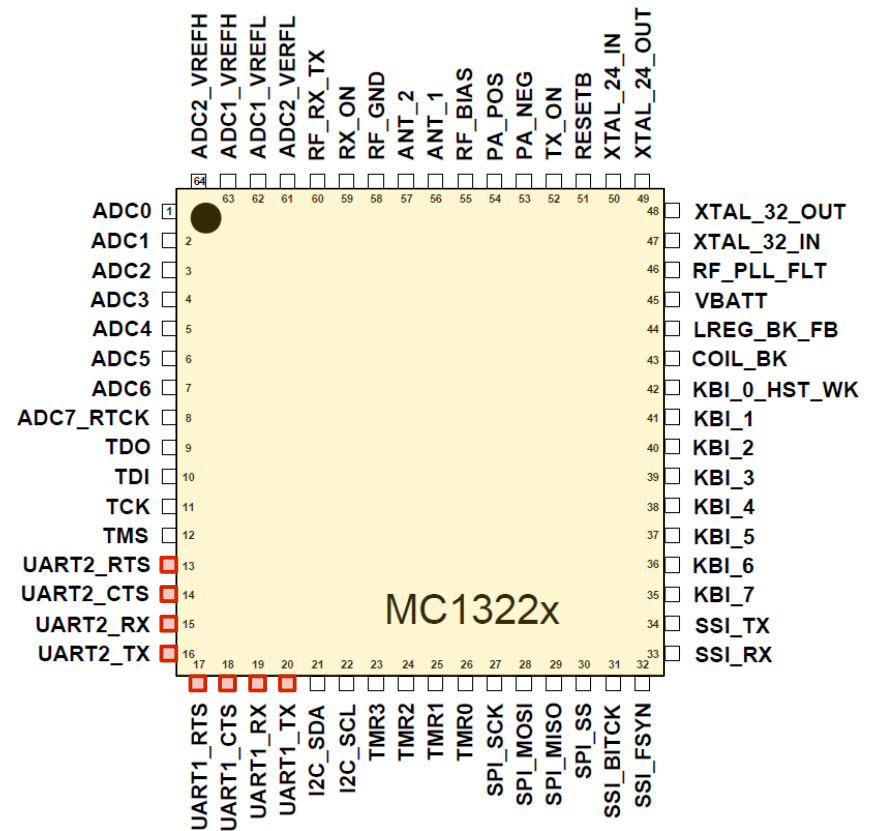
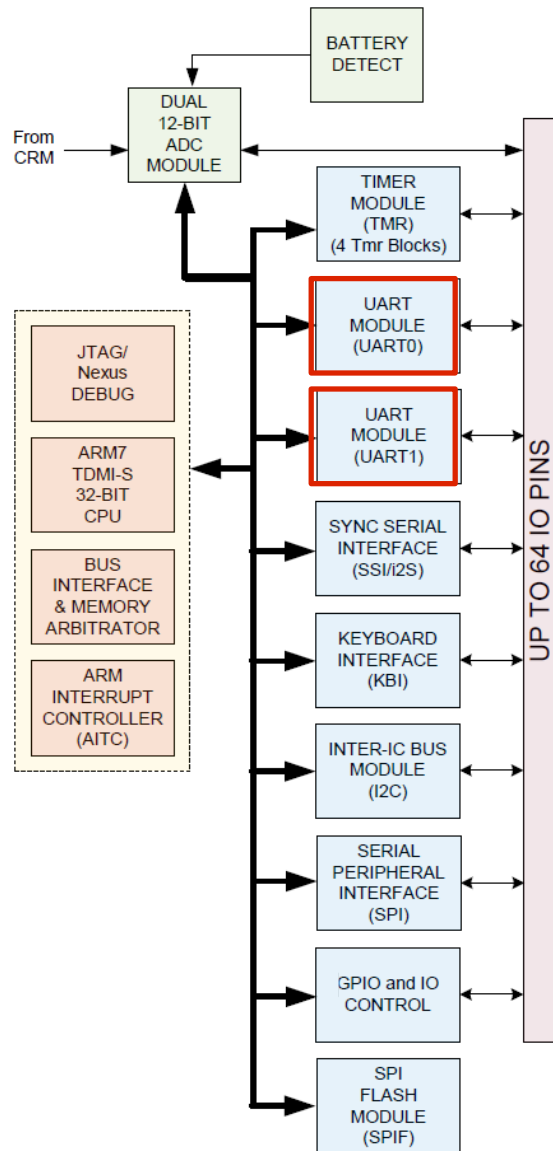
Ejemplo

```
/*
 * Inicializa una uart. Configura los pines del chip y asigna una configuración por defecto (paridad,
 * frecuencia, bits de parada y control de flujo) que no se puede cambiar
 */
void uart_init (uint32_t uart);

/*
 * Transmite un byte por la uart. La llamada a la función se bloquea hasta que transmite el byte
 */
void uart_send_byte (uint32_t uart, uint8_t c);

/*
 * Recibe un byte por la uart. La llamada a la función se bloquea hasta que recibe el byte
 */
uint8_t uart_receive_byte (uint32_t uart);
```

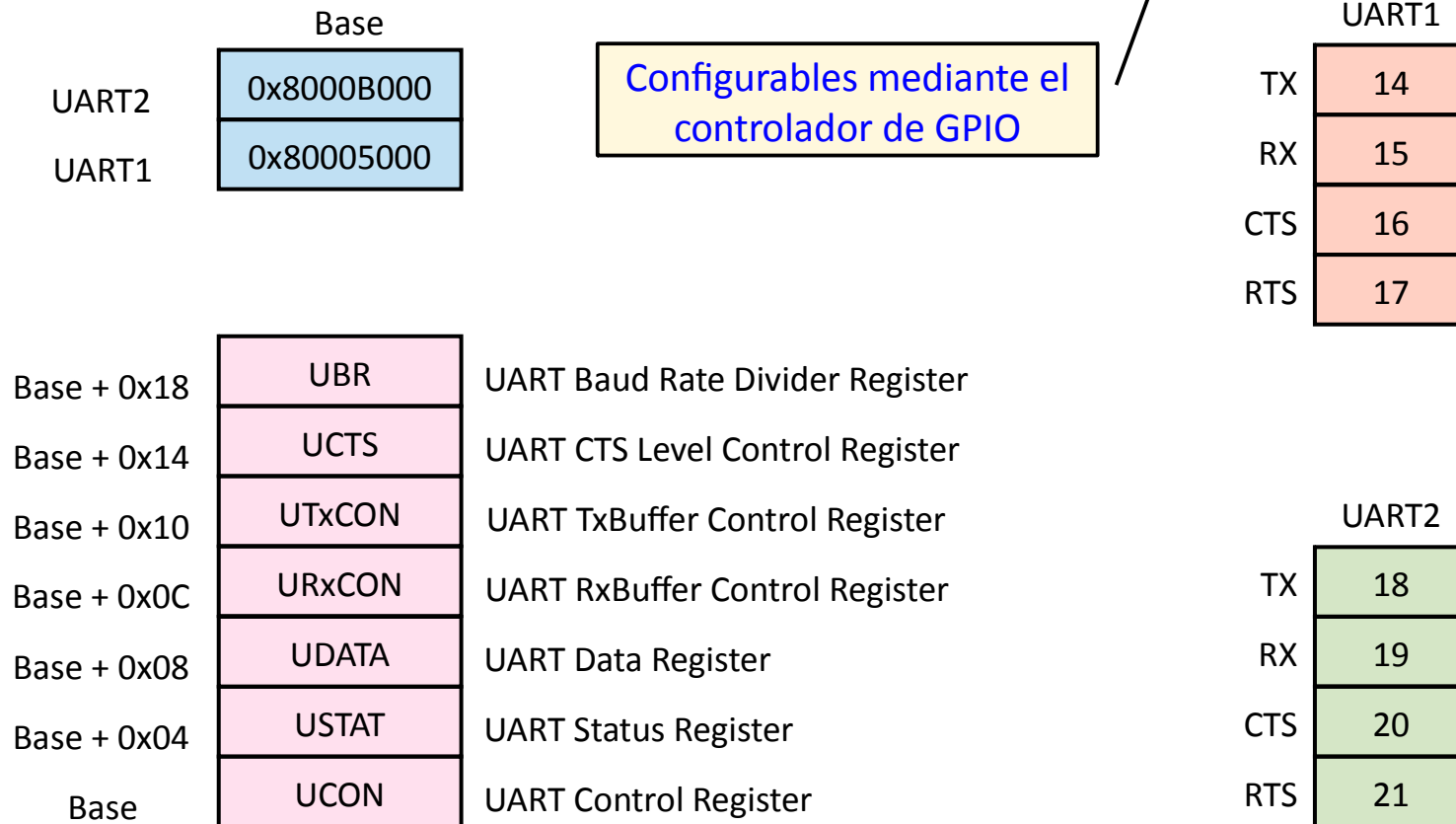
Un driver L0 para las UART del Freescale MC1322x



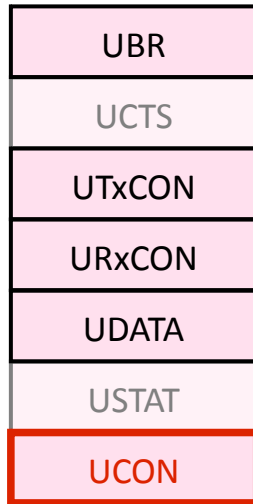
Información fundamental

Mapa de memoria

Asignación de pines



Mapa de memoria del dispositivo




```
typedef struct {
    union {
        struct {
            uint32_t TxE      : 1;
            uint32_t RxE      : 1;
            uint32_t PEN      : 1;
            uint32_t EP       : 1;
            uint32_t ST2      : 1;
            uint32_t SB       : 1;
            uint32_t conTx     : 1;
            uint32_t Tx_oen_b  : 1;
            uint32_t          : 2;
            uint32_t xTIM      : 1;
            uint32_t FCp       : 1;
            uint32_t FCe       : 1;
            uint32_t mTxR      : 1;
            uint32_t mRxR      : 1;
            uint32_t TST       : 1;
        };
        uint32_t CON;
    };
};

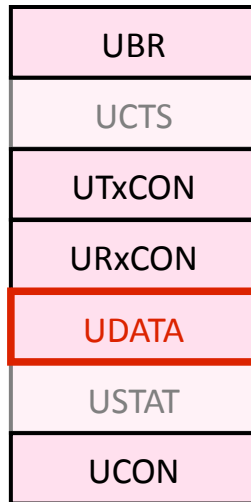
...

} uart_regs_t;
```

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|-----|------|------|-----|-----|------|-----|---|----------|-------|----|-----|----|-----|-----|-----|
| R | TST | mRxr | mTxr | FCe | FCp | xTIM | Res | 0 | Tx_oen_b | conTx | SB | ST2 | EP | PEN | RxE | TxE |
| W | | | | | | | | | | | | | | | | |
| RST | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

 = writes have no effect and terminate without transfer error exception

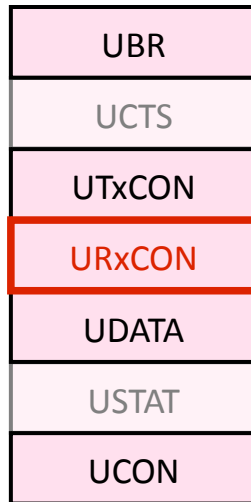
Mapa de memoria del dispositivo



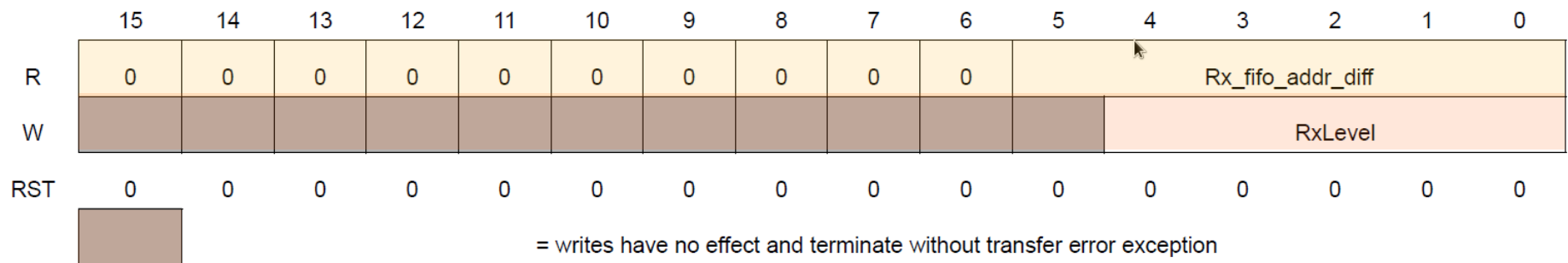
```
typedef struct {  
    :  
    union  
    {  
        uint8_t Rx_data;  
        uint8_t Tx_data;  
        uint32_t DATA;  
    };  
    :  
} uart_regs_t;
```

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|-----------|----|----|----|----|----|---|---|-----------|---|---|---|---|---|---|---|
| R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rx_data | | | | | | | |
| W | | | | | | | | | Tx_Data | | | | | | | |
| RST | Undefined | | | | | | | | Undefined | | | | | | | |

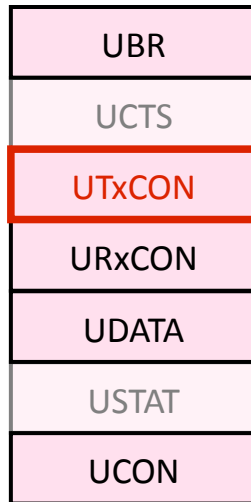
Mapa de memoria del dispositivo



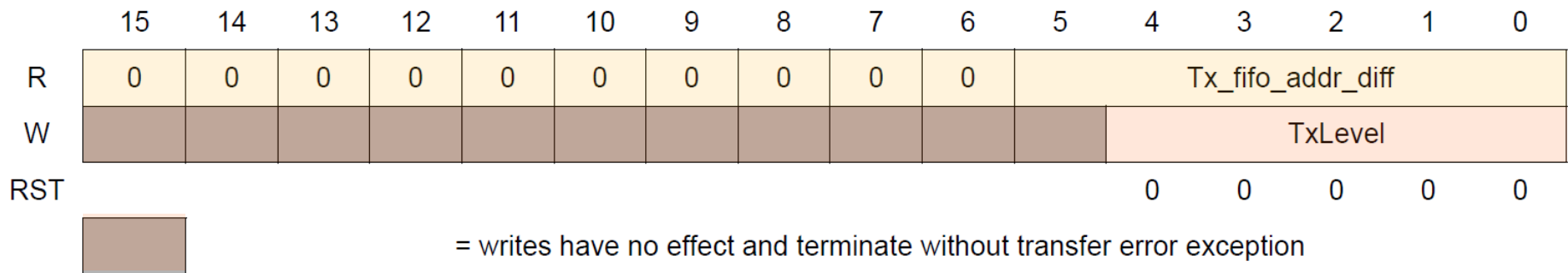
```
typedef struct {
    :
    union
    {
        uint32_t RxLevel          : 5;
        uint32_t Rx_fifo_addr_diff : 6;
        uint32_t RxCON;
    };
    :
} uart_regs_t;
```



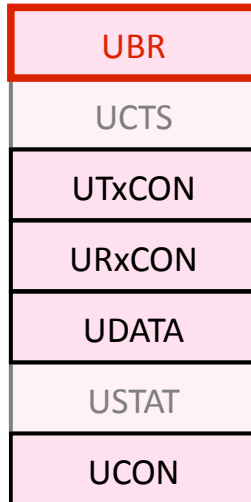
Mapa de memoria del dispositivo



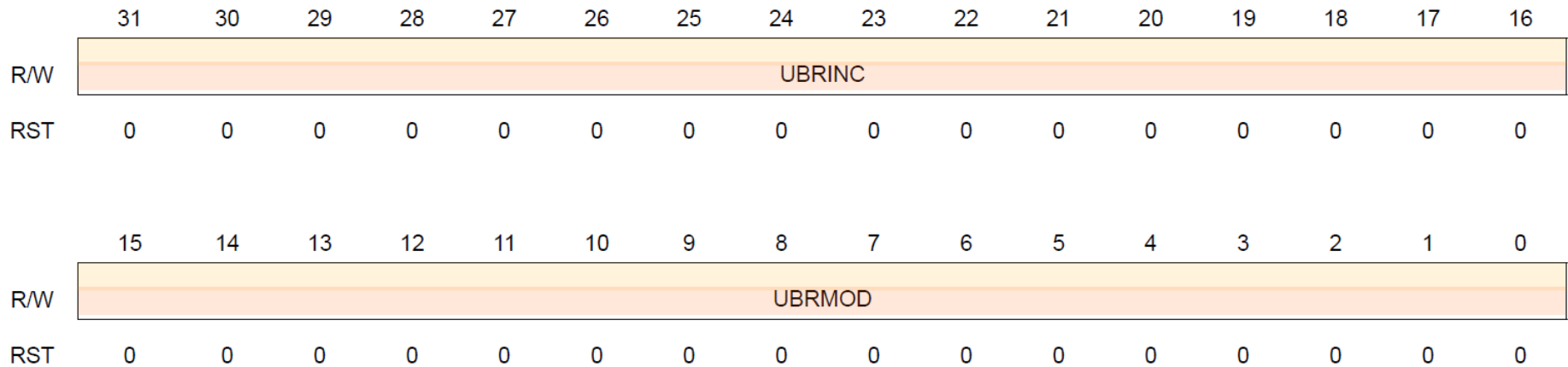
```
typedef struct {
    :
    union
    {
        uint32_t TxLevel           : 5;
        uint32_t Tx_fifo_addr_diff : 6;
        uint32_t TxCON;
    };
    :
} uart_regs_t;
```



Mapa de memoria del dispositivo



```
typedef struct {
    :
    union
    {
        struct
        {
            uint32_t BRMOD : 16;
            uint32_t BRINC : 16;
        };
        uint32_t BR;
    };
} uart_regs_t;
```



Contenidos

Tema 5: Entrada/salida

Introducción

El GPIO

Acceso al mapa de memoria

Diseño de drivers en capas

Drivers de nivel 0

Introducción

Diseño de un driver L0

Drivers de nivel 1

Introducción

Diseño de un driver L1

Drivers de nivel 2

Introducción

Diseño de un driver L2

API del driver L0

bsp/drivers/include/uart.h

```
/**
 * Definición de las uart del sistema
 */
typedef enum {
    uart_0,
    uart_1,
    uart_max
} uart_id_t;

/**
 * Inicializa una uart. Implementación de nivel 0
 * Fija la siguiente configuración por defecto:
 * Tamaño del carácter: 8 bits
 * Baudrate:          115200
 * Paridad:           par
 * Control de flujo:   ninguno
 * Bits de parada:    uno
 * @param uart        Identificador de la uart
 */
void uart_init (uart_id_t uart);

/**
 * Transmite un byte por la uart
 * Implementación del driver de nivel 0. La llamada se bloquea hasta que transmite el byte
 * @param uart        Identificador de la uart
 * @param c           El carácter
 */
void uart_send_byte (uart_id_t uart, uint8_t c);

/**
 * Recibe un byte por la uart
 * Implementación del driver de nivel 0. La llamada se bloquea hasta que recibe el byte
 * @param uart        Identificador de la uart
 * @return            El byte recibido
 */
uint8_t uart_receive_byte (uart_id_t uart);
```

El desarrollador de aplicaciones no tiene por qué conocer el mapa de memoria o la asignación de pines, y no debería tener acceso directo a los registros de control

Gestión del mapa de memoria y de los pines de E/S

bsp/drivers/uart.c

```
/**
 * Acceso estructurado a los registros de control de las uart del MC1322x
 */
typedef struct
{
```

⋮

```
} uart_regs_t;
```

```
/*  
*****  
*/
```

```
/**
 * Acceso estructurado a los pines de las uart del MC1322x
 */
```

```
typedef struct
{
    gpio_pin_t tx, rx, cts, rts;
} uart_pins_t;
```

```
/*  
*****  
*/
```

```
/**
 * Definición de las UARTS
 */
```

```
static volatile uart_regs_t* const uart_regs [uart_max] = {UART1_BASE, UART2_BASE};
```

```
static const uart_pins_t uart_pins [uart_max] = {
    {gpio_pin_14, gpio_pin_15, gpio_pin_16, gpio_pin_17},
    {gpio_pin_18, gpio_pin_19, gpio_pin_20, gpio_pin_21} };
```

Inicialización del dispositivo

Deshabilitamos
la uart

Configuramos los
parámetros

Habilitamos la uart

Asignamos los
pines a la UART

Definimos la
dirección de los pines

bsp/drivers/uart.c

```
/**
 * Inicializa una uart.
 * Implementación de nivel 0
 * Configuración por defecto: 8 bits, 115200 baudios, paridad par,
 * Sin control de flujo, 1 bit de parada
 * @param uart      Identificador de la uart
 */
void uart_init (uart_id_t uart)
{
    uint32_t mod = 9999;
    uint32_t inc = UART_BAUDRATE * mod / (CPU_FREQ >> 4);

    /* Fijamos los parámetros por defecto y deshabilitamos la uart */
    /* La uart debe estar deshabilitada para fijar la frecuencia */
    uart_regs[uart]->CON = (1 << 13) | (1 << 14);

    /* Fijamos la frecuencia, asumimos un oversampling de 8x */
    uart_regs[uart]->BR = ( inc << 16 ) | mod;

    /* Habilitamos la uart. En el MC1322x hay que habilitar el */
    /* periférico antes fijar el modo de funcionamiento de sus pines */
    uart_regs[uart]->CON |= (1 << 0) | (1 << 1);

    /* Cambiamos el modo de funcionamiento de los pines */
    gpio_set_pin_func (uart_pins[uart].tx, gpio_func_alternate_1);
    gpio_set_pin_func (uart_pins[uart].rx, gpio_func_alternate_1);
    gpio_set_pin_func (uart_pins[uart].cts, gpio_func_alternate_1);
    gpio_set_pin_func (uart_pins[uart].rts, gpio_func_alternate_1);

    /* Fijamos TX y CTS como salidas y RX y RTS como entradas */
    gpio_set_pin_dir_output (uart_pins[uart].tx);
    gpio_set_pin_dir_output (uart_pins[uart].cts);
    gpio_set_pin_dir_input  (uart_pins[uart].rx);
    gpio_set_pin_dir_input  (uart_pins[uart].rts);
}
```

Funcionalidad básica

Envío de un carácter

Esperamos a que la UART
esté lista para transmitir

Transmitimos el carácter

bsp/drivers/uart.c

```
/**
 * Transmite un byte por la uart
 * Implementación del driver de nivel 0
 * La llamada se bloquea hasta que transmite el byte
 * @param uart    Identificador de la uart
 * @param c       El carácter
 */
void uart_send_byte (uart_id_t uart, uint8_t c)
{
    /* Esperamos a poder transmitir */
    while (uart_regs[uart]->Tx_fifo_addr_diff == 0);

    /* Escribimos el carácter en la cola HW de la uart */
    uart_regs[uart]->Tx_data = c;
}
```

Recepción de un carácter

Esperamos a que la UART
haya recibido un carácter

Leemos el carácter

bsp/drivers/uart.c

```
/**
 * Recibe un byte por la uart
 * Implementación del driver de nivel 0
 * La llamada se bloquea hasta que recibe el byte
 * @param uart    Identificador de la uart
 * @return        El byte recibido
 */
uint8_t uart_receive_byte (uart_id_t uart)
{
    /* Esperamos a que haya datos que leer */
    while (uart_regs[uart]->Rx_fifo_addr_diff == 0);

    /* Leemos el byte */
    return uart_regs[uart]->Rx_data;
}
```

Contenidos

Tema 5: Entrada/salida

Introducción

El GPIO

Acceso al mapa de memoria

Diseño de drivers en capas

Drivers de nivel 0

Introducción

Diseño de un driver L0

Drivers de nivel 1

Introducción

Diseño de un driver L1

Drivers de nivel 2

Introducción

Diseño de un driver L2

Drivers de nivel 1



Drivers de nivel alto, que permiten un uso flexible y portable del dispositivo

Características:

- API abstracta que aísla a la aplicación de cambios en el hardware
- Soporta la configuración total del dispositivo
- Añade el soporte de interrupciones y funciones callback
- Las llamadas a las funciones del driver no son bloqueantes
- Proporciona interfaces basadas en búferes en vez de en bytes

Ejemplo (a las funciones de nivel 0 se añaden las siguientes)

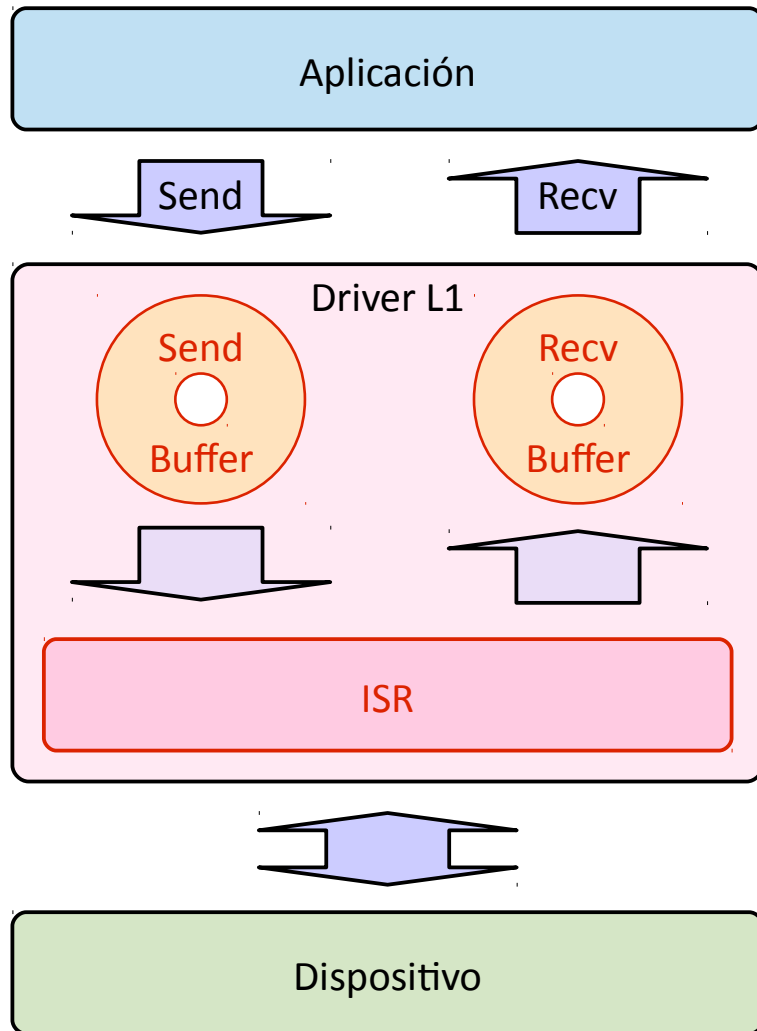
```
ssize_t uart_send (uint32_t uart, char * buf, size_t count);
ssize_t uart_receive (uint32_t uart, char * buf, size_t count);
uart_err_t uart_set_receive_callback (uart_id_t uart, callback_t func);
uart_err_t uart_set_send_callback (uart_id_t uart, callback_t func);
```

E/S

```
uart_err_t uart_set_config (uint32_t uart, uart_config_t params);
uart_err_t uart_get_config (uint32_t uart, uart_config_t * params);
uint32_t uart_is_sending (uint32_t uart);
uint32_t uart_is_receiving (uint32_t uart);
uart_err_t uart_cancel_sending (uint32_t uart);
uart_err_t uart_cancel_receiving (uint32_t uart);
uart_err_t uart_get_status (uint32_t uart);
uart_err_t uart_set_handler (uint32_t uart, handler_t handler);
uart_err_t uart_enable_interrupt (uint32_t uart);
uart_err_t uart_disable_interrupt (uint32_t uart);
```

Gestión del dispositivo

Uso de búferes circulares e interrupciones



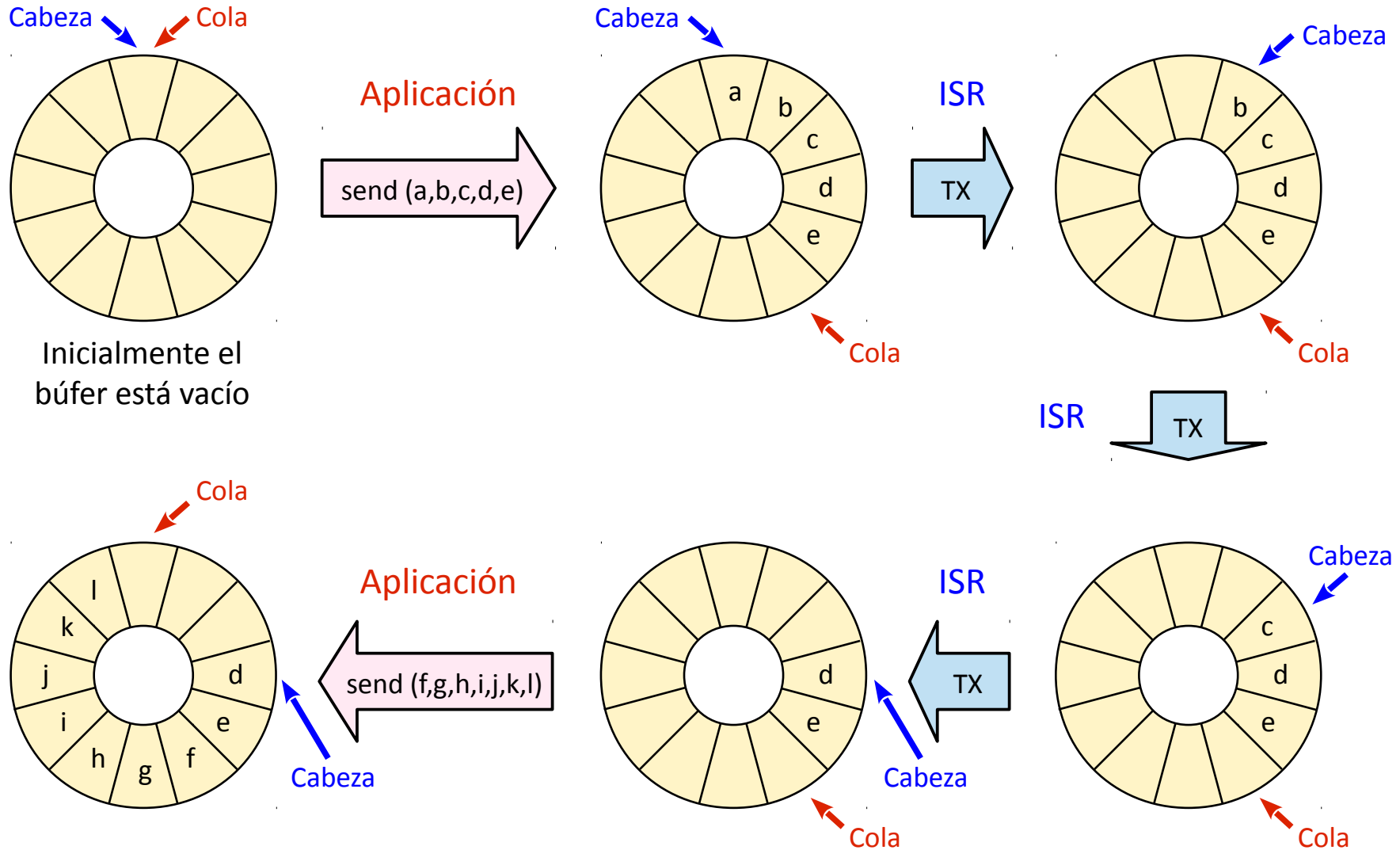
Para la aplicación, una operación de E/S termina en cuanto retorna la función de E/S

Las funciones de E/S simplemente leen o escriben datos en el búfer correspondiente, sin preocuparse de si el dispositivo está disponible, de latencias, anchos de banda, etc.

En cuanto el dispositivo está preparado, interrumpe a la aplicación

La ISR manda o recibe del dispositivo los datos que pueda, sacándolos o insertándolos en el búfer correspondiente, y retorna el control a la aplicación

Ejemplo de funcionamiento: envío de datos al dispositivo



La aplicación inserta los datos en el búfer mediante la función send del driver y sigue ejecutándose

La ISR va sacando los datos del búfer y transmitiéndolos conforme el dispositivo se lo va pidiendo

Funciones *callback*

Utilidad

Son llamadas desde la ISR de un dispositivo a la aplicación para indicarle que han llegado datos o bien que ya se ha terminado un envío

Facilitan la utilización eficiente del dispositivo por parte de la aplicación

Evitan la necesidad de implementar ISRs a la medida de la aplicación (más fácil para el desarrollador)

La ISR es genérica y la funcionalidad de cada aplicación se implementa en las funciones *callback*

Ejemplo de envío de datos

- La aplicación llama a la función *send* del *driver* para ordenar un envío de datos
- La aplicación sigue ejecutándose, despreocupada del envío, mientras que la ISR lo realiza en segundo plano
- Terminada la transferencia, la ISR llama a la función *callback* por si la aplicación quiere enviar más datos

Ejemplo de recepción de datos

- La aplicación se está ejecutando normalmente
- El dispositivo recibe datos e interrumpe a la CPU
- La ISR coloca los datos en el búfer circular del *driver* y llama a la función *callback* de la aplicación
- La función *callback* retira los datos recibidos (llamando a la función *receive*)

Limitación

Las funciones *callback* se ejecutan en el contexto de la ISR, por lo que es muy importante que retornen cuanto antes para no penalizar las prestaciones del sistema

Contenidos

Tema 5: Entrada/salida

Introducción

El GPIO

Acceso al mapa de memoria

Diseño de drivers en capas

Drivers de nivel 0

Introducción

Diseño de un driver L0

Drivers de nivel 1

Introducción

Diseño de un driver L1

Drivers de nivel 2

Introducción

Diseño de un driver L2

API del driver L1

bsp/drivers/include/uart.h

```
/**
 * Definición de las uart del sistema
 */
```

```
typedef enum {
    uart_0,
    uart_1,
    uart_max
} uart_id_t;
```

```
/**
 * Inicializa una uart
 * Fija la siguiente configuración por defecto:
 * @param uart Identificador de la uart
 * @param br Baudrate
 * @return Cero en caso de éxito o -1 en caso de error. La condición de error se indica en la variable global errno
 */
```

```
int32_t uart_init (uart_id_t uart, uint32_t br);
```

```
/**
 * Transmite un byte por la uart
 * Se bloquea hasta que transmite el byte
 * @param uart Identificador de la uart
 * @param c El carácter
 */
```

```
void uart_send_byte (uart_id_t uart, uint8_t c);
```

```
/**
 * Recibe un byte por la uart
 * Se bloquea hasta que recibe el byte
 * @param uart Identificador de la uart
 * @return El byte recibido
 */
```

```
uint8_t uart_receive_byte (uart_id_t uart);
```

L0

```
/**
 * Definición para las funciones de callback
 */
```

```
typedef void (* uart_callback_t) (void);
```

```
/**
 * Transmisión de bytes mediante interrupciones
 * @param uart Identificador de la uart
 * @param buf Búfer con los caracteres
 * @param count Número de caracteres a enviar
 * @return El número de caracteres enviados
 */
```

```
ssize_t uart_send (uint32_t uart, char *buf, size_t count);
```

```
/**
 * Recepción de bytes mediante interrupciones
 * @param uart Identificador de la uart
 * @param buf Búfer para almacenar los bytes
 * @param count Número de caracteres a recibir
 * @return El número de caracteres recibidos
 */
```

```
ssize_t uart_receive (uint32_t uart, char *buf, size_t count);
```

```
/**
 * Fija la función callback de recepción de una uart
 * @param uart Identificador de la uart
 * @param func Función callback
 */
```

```
void uart_set_receive_callback (uart_id_t uart, uart_callback_t func);
```

```
/**
 * Fija la función callback de transmisión de una uart
 * @param uart Identificador de la uart
 * @param func Función callback
 */
```

```
void uart_set_send_callback (uart_id_t uart, uart_callback_t func);
```

L1

Gestión del mapa de memoria y de los pines de E/S

bsp/drivers/uart.c

```
/**
 * Acceso estructurado a los registros de control
 * de las uart del MC1322x
 */
typedef struct
{
    :

} uart_regs_t;

/*****/

/**
 * Acceso estructurado a los pines de las uart del MC1322x
 */
typedef struct
{
    gpio_pin_t tx, rx, cts, rts;
} uart_pins_t;

/*****/

/**
 * Definición de las UARTS
 */
static volatile uart_regs_t* const uart_regs [uart_max]
    = {UART1_BASE, UART2_BASE};

static const uart_pins_t uart_pins [uart_max] = {
    {gpio_pin_14, gpio_pin_15, gpio_pin_16, gpio_pin_17},
    {gpio_pin_18, gpio_pin_19, gpio_pin_20, gpio_pin_21} };
```

L0

```
/**
 * Acceso estructurado a las ISR
 */
static void uart_1_isr (void);
static void uart_2_isr (void);
static const itc_handler_t uart_irq_handlers[uart_max] =
    {uart_1_isr, uart_2_isr};

/**
 * Búferes circulares
 */
#include "circular_buffer.h"
#define __UART_BUFFER_SIZE__ 256

static volatile uint8_t
    uart_rx_buffers[uart_max][__UART_BUFFER_SIZE__];
static volatile uint8_t
    uart_tx_buffers[uart_max][__UART_BUFFER_SIZE__];

static volatile circular_buffer_t
    uart_circular_rx_buffers[uart_max];
static volatile circular_buffer_t
    uart_circular_tx_buffers[uart_max];

/**
 * Gestión de las callbacks
 */
typedef struct
{
    uart_callback_t tx_callback;
    uart_callback_t rx_callback;
} uart_callbacks_t;

static volatile uart_callbacks_t uart_callbacks[uart_max];
```

L1

Inicialización del dispositivo

bsp/drivers/uart.c

```
/**
 * Inicializa una uart. Implementación de nivel 1
 * @param uart Identificador de la uart
 * @param br Baudrate
 * @return Cero en caso de éxito o -1 en caso de error.
 * La condición de error se indica en la
 * variable global errno L1
 */
int32_t uart_init (uart_id_t uart, uint32_t br)
{
    uint32_t mod = 9999;
    uint32_t inc = br * mod / (CPU_FREQ >> 4);

    /* Comprobación de errores */ L1
    if (uart >= uart_max) {
        errno = ENODEV; /* El dispositivo no existe */
        return -1;
    }

    /* Fijamos los parámetros por defecto */

    /* La uart debe estar deshabilitada para fijar la */
    /* frecuencia. Anulamos la generación de interrupciones */
    uart_regs[uart]->CON = (1 << 13) | (1 << 14);

    /* Fijamos la frecuencia, asumimos oversampling de 8x */
    uart_regs[uart]->BR = ( inc << 16 ) | mod;

    /* En el MC1322x hay que habilitar el periférico antes */
    /* de fijar el modo de funcionamiento de sus pines */
    uart_regs[uart]->CON |= (1 << 0) | (1 << 1);

    /* Cambiamos el modo de funcionamiento de los pines */
    gpio_set_pin_func(uart_pins[uart].tx, gpio_func_alternate_1);
    gpio_set_pin_func(uart_pins[uart].rx, gpio_func_alternate_1);
    gpio_set_pin_func(uart_pins[uart].cts, gpio_func_alternate_1);
    gpio_set_pin_func(uart_pins[uart].rts, gpio_func_alternate_1);
}
```

```
/* Fijamos la dirección de los pines */
gpio_set_pin_dir_output (uart_pins[uart].tx);
gpio_set_pin_dir_output (uart_pins[uart].cts);
gpio_set_pin_dir_input (uart_pins[uart].rx);
gpio_set_pin_dir_input (uart_pins[uart].rts);

/* Inicializamos los búferes de circulares */ L1
circular_buffer_init (
    &uart_circular_rx_buffers[uart],
    (uint8_t *) uart_rx_buffers[uart],
    sizeof(uart_rx_buffers[uart]));
circular_buffer_init (
    &uart_circular_tx_buffers[uart],
    (uint8_t *) uart_tx_buffers[uart],
    sizeof(uart_tx_buffers[uart]));

/* Programamos cuando generar las interrupciones */
uart_regs[uart]->TxLevel = 31; /* cola envío vacía */
uart_regs[uart]->RxLevel = 1; /* llega un byte */

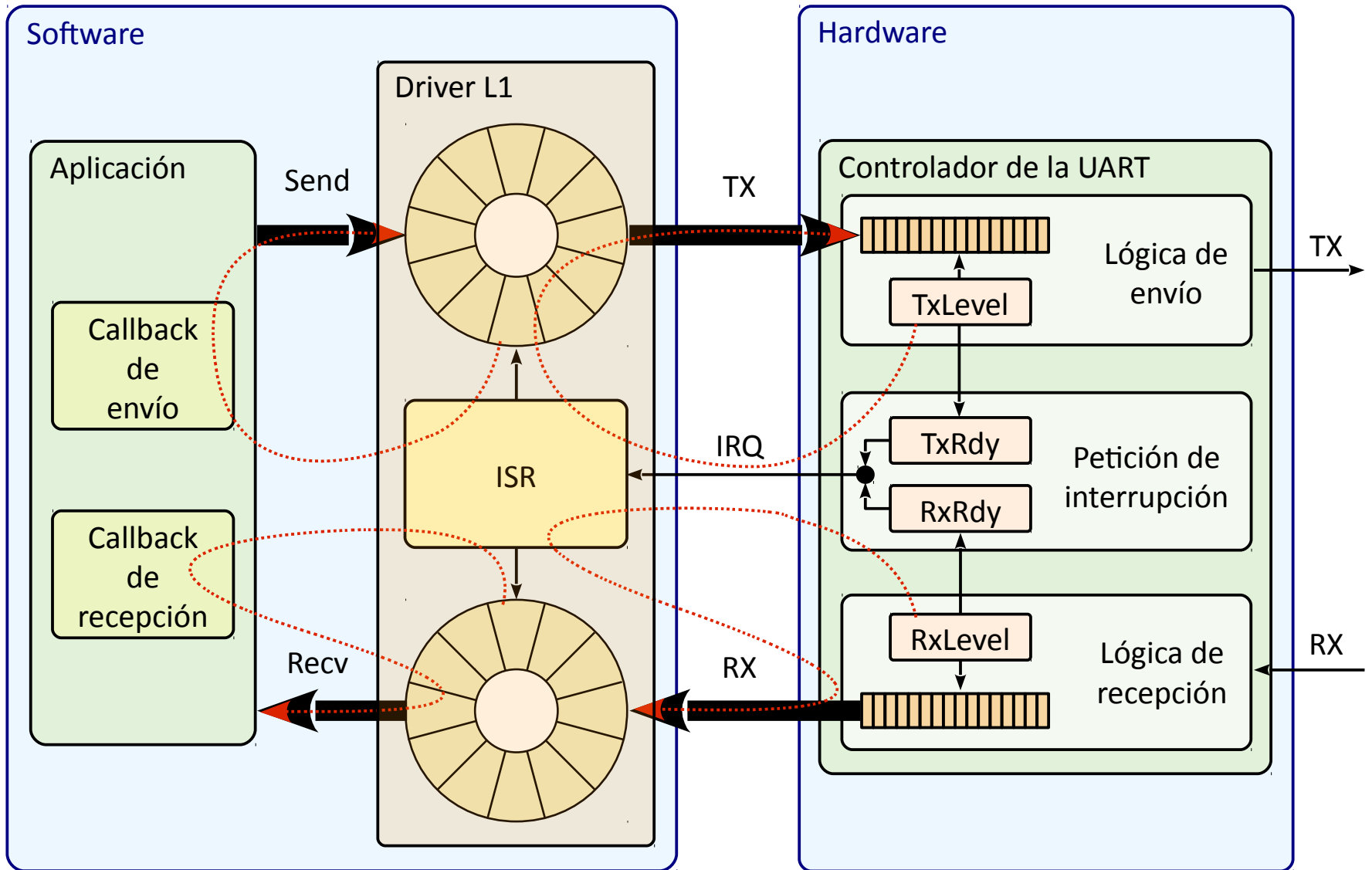
/* Habilitamos las interrupciones de la uart */
/* en el controlador de interrupciones del sistema */
itc_set_priority (itc_src_uart1 + uart,
    itc_priority_normal);
itc_set_handler (itc_src_uart1 + uart,
    uart_irq_handlers[uart]);
itc_enable_interrupt (itc_src_uart1 + uart);

/* Por defecto no hay funciones callback */
uart_callbacks[uart].tx_callback = NULL;
uart_callbacks[uart].rx_callback = NULL;

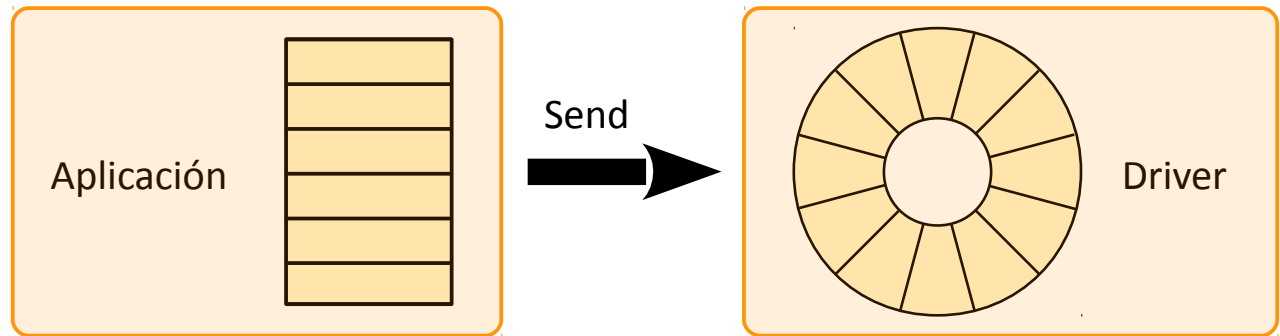
/* Habilitamos interrupciones en la recepción */
uart_regs[uart]->mRxR = 0;

return 0;
}
```


Envío y recepción mediante búferes e interrupciones



Envío basado en búferes



bsp/drivers/uart.c

```
/**
 * Transmisión de bytes. Implementación del driver de nivel 1
 * @param uart    Identificador de la uart
 * @param buf     Búfer con los caracteres
 * @param count   Número de caracteres a escribir
 * @return        El número de bytes almacenados en el búfer de transmisión
 */
```

```
ssize_t uart_send (uint32_t uart, char *buf, size_t count)
{
    ssize_t buffered_count = 0;
```

```
    /* Enmascaramos las interrupciones del transmisor para que no nos interfiera la ISR */
    uart_regs[uart]->mTxR = 1;    /* Comienzo de región crítica */
```

```
    /* Insertamos en el búfer circular los bytes que podamos */
    while (!circular_buffer_is_full (&uart_circular_tx_buffers[uart]) && count > 0)
    {
        circular_buffer_write (&uart_circular_tx_buffers[uart], *buf++);
        buffered_count++;
        count--;
    }
```

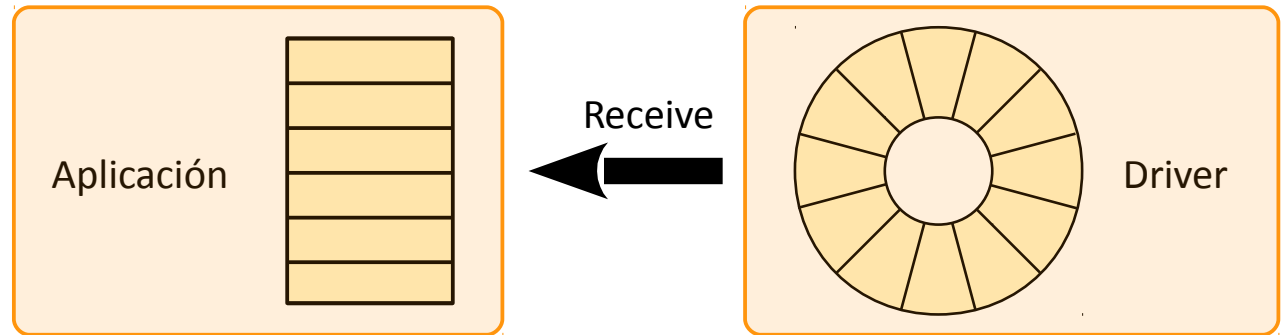
Región crítica

```
    /* Desenmascaramos las interrupciones del transmisor */
    uart_regs[uart]->mTxR = 0;    /* Fin de región crítica */
```

```
    return buffered_count;
```

```
}
```

Recepción basada en búferes



bsp/drivers/uart.c

```
/**
 * Recepción de bytes. Implementación del driver de nivel 1
 * @param uart   Identificador de la uart
 * @param buf     Búfer para almacenar los bytes
 * @param count   Número de bytes a leer
 * @return        El número de bytes realmente leídos y almacenados en el búfer circular
 */
```

```
ssize_t uart_receive (uint32_t uart, char *buf, size_t count)
{
    ssize_t buffered_count = 0;
```

```
/* Enmascaramos las interrupciones del receptor para que no nos interfiera la ISR */
uart_regs[uart]->mRxR = 1; /* Comienzo de sección crítica */
```

```
/* Leemos del búffer circular los bytes que podamos */
while (!circular_buffer_is_empty (&uart_circular_rx_buffers[uart]) && count > 0)
{
    *buf++ = circular_buffer_read (&uart_circular_rx_buffers[uart]);
    buffered_count++;
    count--;
}
```

Región crítica

```
/* Desenmascaramos las interrupciones del receptor */
uart_regs[uart]->mRxR = 0; /* Fin de sección crítica */
```

```
return buffered_count;
```

```
}
```

ISRs

bsp/drivers/uart.c

```
/**
 * Cada isr llamará a este manejador genérico. Lo declaramos inline para reducir la latencia de la isr
 * @param uart    Identificador de la uart
 */
static inline void uart_isr (uart_id_t uart) {
    uint32_t status = uart_regs[uart]->STAT;          /* Limpiamos los bits de error, de momento no gestionamos errores */

    if (uart_regs[uart]->RxRdy) {                      /* Si la interrupción es del receptor */
        /* Mandamos al búfer todos los caracteres de la cola HW que podamos */
        while (!circular_buffer_is_full(&uart_circular_rx_buffers[uart]) && (uart_regs[uart]->Rx_fifo_addr_diff > 0))
            circular_buffer_write(&uart_circular_rx_buffers[uart], uart_regs[uart]->Rx_data); /* Recibimos un carácter */

        /* Llamamos a la función callback para que la aplicación se haga cargo de los datos del búfer */
        if (uart_callbacks[uart].rx_callback) uart_callbacks[uart].rx_callback();

        /* Si el buffer circular está lleno, no podemos aceptar más datos */
        if (circular_buffer_is_full (&uart_circular_rx_buffers[uart]))
            uart_regs[uart]->mRxR = 1; /* Enmascaramos las interrupciones del receptor para que no nos ofrezca más datos */
    }

    if (uart_regs[uart]->TxRdy) {                      /* Si la interrupción es del transmisor */
        /* Mandamos a la cola HW todos los caracteres del búfer que podamos */
        while (!circular_buffer_is_empty(&uart_circular_tx_buffers[uart]) && (uart_regs[uart]->Tx_fifo_addr_diff > 0))
            uart_regs[uart]->Tx_data = circular_buffer_read (&uart_circular_tx_buffers[uart]); /* Transmitimos un carácter */

        /* Llamamos a la función callback por si la aplicación quiere mandar más datos al búfer */
        if (uart_callbacks[uart].tx_callback) uart_callbacks[uart].tx_callback();

        /* Si el búfer está vacío es que no hay mas datos */
        if (circular_buffer_is_empty (&uart_circular_tx_buffers[uart]))
            uart_regs[uart]->mTxR = 1; /* Enmascaramos las interrupciones del transmisor para que no nos pida más datos */
    }
}

/** Manejadores de interrupciones (ISR) para las uart */
static void uart_1_isr (void) { uart_isr(uart_1); }
static void uart_2_isr (void) { uart_isr(uart_2); }
```