Sistemas Empotrados

Tema 5:
Entrada/salida

Lección 12:
Introducción al diseño de drivers de E/S

Jesús González Peñalver

Contenidos

Gestión de los pines de un SoC

Periféricos internos

Un SoC dispone de multitud de periféricos on-chip (UART, controlador USB, temporizadores, controlador LCD, etc.) que necesitan hacer uso de pines para poder comunicarse con el exterior

E/S de propósito general

Por otro lado, es necesario disponer de pines de E/S de propósito general para poder conectar al SoC otros dispositivos externos (sensores, actuadores, leds, botones, memorias, etc.)

Problema

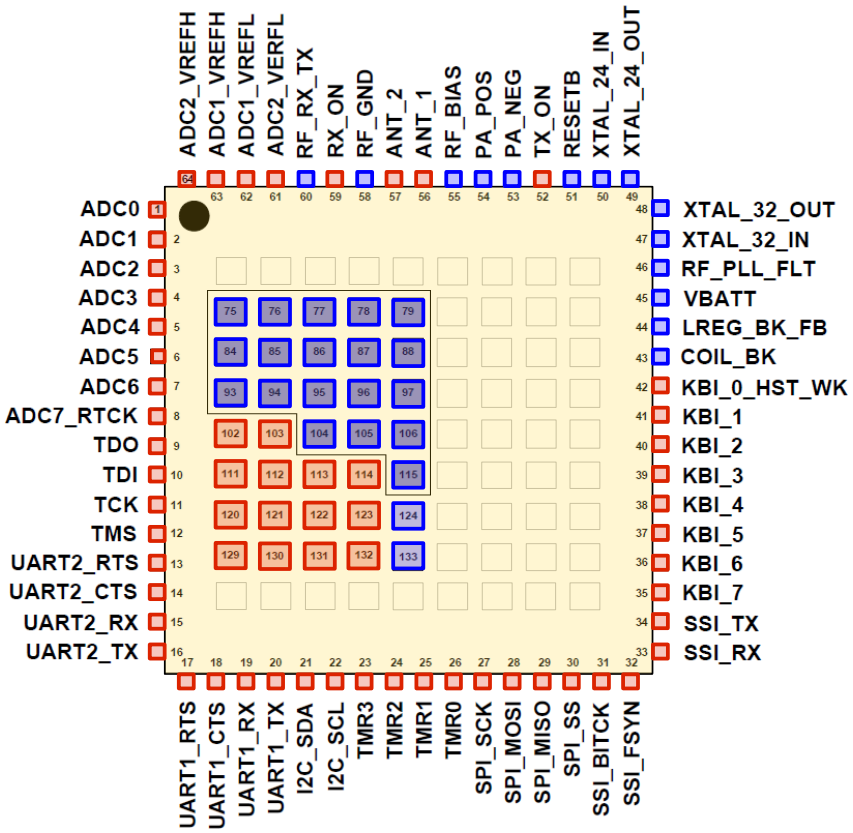
Cuantos más pines tenga el SoC, mayor será su coste

Solución: Multiplexación de las pines de entrada/salida

La mayoría de los pines están compartidos entre los periféricos internos y la E/S de propósito general

El controlador GPIO (General Purpose I/O) gestiona si cada pin es de entrada, de salida, si está conectado a un dispositivo interno, o si se usará como E/S de propósito general

Ejemplo: Freescale MC1322x



Número total de pines:	145
Número de pines sin conectar:	46 (robustez ante golpes)
Número de pines multiplexados:	64 (GPIO0 - GPIO63)
Número de pines de uso específico:	35

Fuente: MC1322x Advanced ZigBee™-Compliant SoC Platform for the 2.4 GHz IEEE® 802.15.4 Standard Reference Manual
Freescale Semiconductor. Document Number MC1322xRM

El controlador GPIO

Gestiona el uso de los pines de E/S

Si cada pin es de entrada, de salida, si está conectado a un dispositivo interno, o si se usará como E/S de propósito general (GPIO)

Agrupa los pines en puertos de E/S

Permite configurar y acceder a todos los pines de un puerto en paralelo

Ejemplo: Freescale MC1322x

64 GPIOs agrupados en dos puertos de 32 bits

Port 0						Port 1					
Bit	Function	Pin	Bit	Function	Pin	Bit	Function	Pin	Bit	Function	Pin
0	GPIO0 / SSI_TX	34	16	GPIO16 / UART1_CTS	18	0	GPIO32 / ADC2	3	16	GPIO48 / TDI	10
1	GPIO1 / SSI_RX	33	17	GPIO17 / UART1_RTS	17	1	GPIO33 / ADC3	4	17	GPIO49 / TDO	9
2	GPIO2 / SSI_FSYN	32	18	GPIO18 / UART2_TX	16	2	GPIO34 / ADC4	5	18	GPIO50 / MCKO	131
3	GPIO3 / SSI_BITCK	31	19	GPIO19 / UART2_RX	15	3	GPIO35 / ADC5	6	19	GPIO51 / MDO00	103
4	GPIO4 / SPI_SS	30	20	GPIO20 / UART2_CTS	14	4	GPIO36 / ADC6	7	20	GPIO52 / MDO01	102
5	GPIO5 / SPI_MISO	29	21	GPIO21 / UART2_RTS	13	5	GPIO37 / ADC7_RTCK	8	21	GPIO53 / MDO02	112
6	GPIO6 / SPI_MOSI	28	22	GPIO22 / KBI_0_HST_WK	42	6	GPIO38 / ADC2_VREFH	64	22	GPIO54 / MDO03	111
7	GPIO7 / SPI_SCK	27	23	GPIO23 / KBI_1	41	7	GPIO39 / ADC2_VREFL	61	23	GPIO55 / MDO04	121
8	GPIO8 / TMR0	26	24	GPIO24 / KBI_2	40	8	GPIO40 / ADC1_VREFH	63	24	GPIO56 / MDO07	120
9	GPIO9 / TMR1	25	25	GPIO25 / KBI_3	39	9	GPIO41 / ADC1_VREFL	62	25	GPIO57 / MDO06	130
10	GPIO10 / TMR2	24	26	GPIO26 / KBI_4	38	10	GPIO42 / ANT_1	56	26	GPIO58 / MDO07	129
11	GPIO11 / TMR3	23	27	GPIO27 / KBI_5	37	11	GPIO43 / ANT_2	57	27	GPIO59 / MSEO0_B	114
12	GPIO12 / I2C_SCL	22	28	GPIO28 / KBI_6	36	12	GPIO44 / TX_ON	52	28	GPIO60 / MSEO1_B	113
13	GPIO13 / I2C_SDA	21	29	GPIO29 / KBI_7	35	13	GPIO45 / RX_ON	59	29	GPIO61 / RDY_B	122
14	GPIO14 / UART1_TX	20	30	GPIO30 / ADC_0	1	14	GPIO46 / TMS	12	30	GPIO62 / EVTO_B	123
15	GPIO15 / UART1_RX	19	31	GPIO31 / ADC_1	2	15	GPIO47 / TCK	11	31	GPIO63 / EVTI_B	132

Fuente: MC1322x Advanced ZigBee™-Compliant SoC Platform for the 2.4 GHz IEEE® 802.15.4 Standard Reference Manual Freescale Semiconductor. Document Number MC1322xRM

Gestión del controlador GPIO

Interfaz

Como todos los dispositivos *on-chip* la gestión del GPIO se realiza mediante un conjunto de registros de control/estado mapeados en la RAM del SoC

Ejemplo: Freescale MC1322x

Dirección base: 0x8000 0000

Cada pin puede tener hasta 4 modos de funcionamiento diferentes

Address	Name
Base + 0x00	GPIO Pad Direction for GPIO 00-31 (GPIO_PAD_DIR0)
Base + 0x04	GPIO Pad Direction for GPIO 32-63 (GPIO_PAD_DIR1)
Base + 0x08	GPIO Data for GPIO 00-31 (GPIO_DATA0)
Base + 0x0C	GPIO Data for GPIO 32-63 (GPIO_DATA1)
Base + 0x10	GPIO Pad Pull-up Enable for GPIO 00-31 (GPIO_PAD_PU_EN0)
Base + 0x14	GPIO Pad Pull-up Enable for GPIO 32-63 (GPIO_PAD_PU_EN1)
Base + 0x18	GPIO Function Select for GPIO 00-15 (GPIO_FUNC_SEL0)
Base + 0x1C	GPIO Function Select for GPIO 16-31 (GPIO_FUNC_SEL1)
Base + 0x20	GPIO Function Select for GPIO 32-47 (GPIO_FUNC_SEL2)
Base + 0x24	GPIO Function Select for GPIO 48-63 (GPIO_FUNC_SEL3)
Base + 0x28	GPIO Data Select for GPIO 00-31 (GPIO_DATA_SEL0)
Base + 0x2C	GPIO Data Select for GPIO 32-63 (GPIO_DATA_SEL1)
Base + 0x30	GPIO Pad Pull-up Select for GPIO 00-31 (GPIO_PAD_PU_SEL0)

Address	Name
Base + 0x34	GPIO Pad Pull-up Select for GPIO 32-63 (GPIO_PAD_PU_SEL1)
Base + 0x38	GPIO Pad Hysteresis Enable for GPIO 00-31 (GPIO_PAD_HYST_EN0)
Base + 0x3C	GPIO Pad Hysteresis Enable for GPIO 32-63 (GPIO_PAD_HYST_EN1)
Base + 0x40	GPIO Pad Keeper Enable for GPIO 00-31 (GPIO_PAD_KEEP0)
Base + 0x44	GPIO Pad Keeper Enable for GPIO 32-63 (GPIO_PAD_KEEP1)
Base + 0x48	GPIO Data Set for GPIO 00-31 (GPIO_DATA_SET0)
Base + 0x4C	GPIO Data Set for GPIO 32-63 (GPIO_DATA_SET1)
Base + 0x50	GPIO Data Reset for GPIO 00-31 (GPIO_DATA_RESET0)
Base + 0x54	GPIO Data Reset for GPIO 32-63 (GPIO_DATA_RESET1)
Base + 0x58	GPIO Pad Direction Set for GPIO 00-31 (GPIO_PAD_DIR_SET0)
Base + 0x5C	GPIO Pad Direction Set for GPIO 32-63 (GPIO_PAD_DIR_SET1)
Base + 0x60	GPIO Pad Direction Reset for GPIO 00-31 (GPIO_PAD_DIR_RESET0)
Base + 0x64	GPIO Pad Direction Reset for GPIO 32-63 (GPIO_PAD_DIR_RESET1)

Fuente: MC1322x Advanced ZigBee™-Compliant SoC Platform for the 2.4 GHz IEEE® 802.15.4 Standard Reference Manual Freescale Semiconductor. Document Number MC1322xRM

Acceso estructurado a los registros de control del GPIO

bsp/drivers/gpio.c

```
#include <stdint.h>
#include "system.h"

typedef struct
{
    uint32_t  PAD_DIR[2];       /* Selección de dirección del pad */
    uint32_t  DATA[2];          /* Datos */
    uint32_t  PU_EN[2];         /* Habilitación de pull-up/pull_down */
    uint32_t  FUNC_SEL[4];      /* Selección del modo de funcionamiento de cada pin */
    uint32_t  DATA_SEL[2];      /* Selección del origen de los datos de entrada */
    uint32_t  PAD_PU_SEL[2];    /* Selección entre pull-up y pull-down */
    uint32_t  PAD_HYST_EN[2];   /* Habilitación de histéresis */
    uint32_t  PAD_KEEP[2];      /* Habilitación de mantener el último estado del pin */
    uint32_t  DATA_SET[2];      /* Activa bits en GPIO_DATA */
    uint32_t  DATA_RESET[2];    /* Desactiva bits en GPIO_DATA */
    uint32_t  DIR_SET[2];       /* Activa bits en GPIO_PAD_DIR */
    uint32_t  DIR_RESET[2];     /* Desactiva bits en GPIO_PAD_DIR */
} gpio_regs_t;

static volatile gpio_regs_t* const gpio_regs = GPIO_BASE;
```

Nos aseguramos que los campos son de 32 bits (uint32_t está definido en stdint.h)

gpio_regs sólo es accesible desde este fichero

No se permite modificar el puntero gpio_regs

Definido en system.h (0x8000 0000)

El contenido apuntado por el puntero gpio_regs puede variar sin que lo modifique el programa

API para la gestión del GPIO

bsp/drivers/include/gpio.h

```
/**
 * Definición de los puertos del GPIO
 */
typedef enum {
     gpio_port_0,
     gpio_port_1,
     gpio_port_max
} gpio_port_t;

/**
 * Definición de los diferentes modos de funcionnamiento para cada pin del GPIO
 */
typedef enum
{
     gpio_func_normal,
     gpio_func_alternate_1,
     gpio_func_alternate_2,
     gpio_func_alternate_3,
     gpio_func_max
} gpio_func_t;

/**
 * Definición de los pines del GPIO
 */
typedef enum
{
     gpio_pin_0,  gpio_pin_1,  gpio_pin_2,  gpio_pin_3,  gpio_pin_4,  gpio_pin_5,  gpio_pin_6,  gpio_pin_7,
     gpio_pin_8,  gpio_pin_9,  gpio_pin_10, gpio_pin_11, gpio_pin_12, gpio_pin_13, gpio_pin_14, gpio_pin_15,
     gpio_pin_16, gpio_pin_17, gpio_pin_18, gpio_pin_19, gpio_pin_20, gpio_pin_21, gpio_pin_22, gpio_pin_23,
     gpio_pin_24, gpio_pin_25, gpio_pin_26, gpio_pin_27, gpio_pin_28, gpio_pin_29, gpio_pin_30, gpio_pin_31,
     gpio_pin_32, gpio_pin_33, gpio_pin_34, gpio_pin_35, gpio_pin_36, gpio_pin_37, gpio_pin_38, gpio_pin_39,
     gpio_pin_40, gpio_pin_41, gpio_pin_42, gpio_pin_43, gpio_pin_44, gpio_pin_45, gpio_pin_46, gpio_pin_47,
     gpio_pin_48, gpio_pin_49, gpio_pin_50, gpio_pin_51, gpio_pin_52, gpio_pin_53, gpio_pin_54, gpio_pin_55,
     gpio_pin_56, gpio_pin_57, gpio_pin_58, gpio_pin_59, gpio_pin_60, gpio_pin_61, gpio_pin_62, gpio_pin_63,
     gpio_pin_max
} gpio_pin_t;
```

Es importante dejar claro en el API qué parámetros se pueden usar para llamar a las funciones del driver

Para ello se deben definir tantos tipos de dato como sea necesario

API para la gestión del GPIO

bsp/drivers/include/gpio.h

Gestión del GPIO a nivel de puertos

```
/**
 * Fija la dirección los pines seleccionados en la máscara como de entrada
 * @param    port         Puerto
 * @param    mask         Máscara para seleccionar los pines
 */
void gpio_set_port_dir_input (gpio_port_t port, uint32_t mask);

/**
 * Fija la dirección los pines seleccionados en la máscara como de salida
 * @param    port         Puerto
 * @param    mask         Máscara para seleccionar los pines
 */
void gpio_set_port_dir_output (gpio_port_t port, uint32_t mask);

/**
 * Escribe unos en los pines seleccionados en la máscara
 * @param    port         Puerto
 * @param    mask         Máscara para seleccionar los pines
 */
void gpio_set_port (gpio_port_t port, uint32_t mask);

/**
 * Escribe ceros en los pines seleccionados en la máscara
 * @param    port         Puerto
 * @param    mask         Máscara para seleccionar los pines
 */
void gpio_clear_port (gpio_port_t port, uint32_t mask);

/**
 * Asigna un modo de funcionamiento a los pines seleccionados por la máscara
 * @param    port         Puerto
 * @param    func         Modo de funcionamiento
 * @param    mask         Máscara para seleccionar los pines
 */
void gpio_set_port_func (gpio_port_t port, gpio_func_t func, uint32_t mask);
```

API para la gestión del GPIO

bsp/drivers/include/gpio.h

Gestión del GPIO a nivel de pines

```
/**
 * Fija la dirección del pin indicado como de entrada
 *
 * @param    pin          Número de pin
 */
void gpio_set_pin_dir_input (gpio_pin_t pin);

/**
 * Fija la dirección del pin indicado como de salida
 *
 * @param    pin          Número de pin
 */
void gpio_set_pin_dir_output (gpio_pin_t pin);

/**
 * Escribe un uno en el pin indicado
 *
 * @param    pin          Número de pin
 */
void gpio_set_pin (gpio_pin_t pin);

/**
 * Escribe un cero en el pin indicado
 *
 * @param    pin          Número de pin
 */
void gpio_clear_pin (gpio_pin_t pin);

/**
 * Asigna un modo de funcionamiento al pin seleccionado
 *
 * @param    pin          Pin
 * @param    func         Modo de funcionamiento
 */
void gpio_set_pin_func (gpio_pin_t pin, gpio_func_t func);
```

API para la gestión del GPIO – Ejemplos

bsp/drivers/gpio.c

```
/**
 * Fija la dirección los pines seleccionados en la máscara como de entrada
 * @param    port         Puerto
 * @param    mask         Máscara para seleccionar los pines
 */
inline void gpio_set_port_dir_input (gpio_port_t port, uint32_t mask)
{
            gpio_regs->DIR_RESET[port] = mask;
}

/**
 * Escribe ceros en los pines seleccionados en la máscara
 * @param    port         Puerto
 * @param    mask         Máscara para seleccionar los pines
 */
inline void gpio_clear_port (gpio_port_t port, uint32_t mask)
{
            gpio_regs->DATA_RESET[port] = mask;
}

/**
 * Fija la dirección del pin indicado como de salida
 * @param    pin          Número de pin
 */
inline void gpio_set_pin_dir_output (gpio_pin_t pin)
{
            gpio_regs->DIR_SET[(pin >> 5) & 1] = 1 << (pin & 0x1f);
}

/**
 * Escribe un uno en el pin indicado
 * @param    pin          Número de pin
 */
inline void gpio_set_pin (gpio_pin_t pin)
{
            gpio_regs->DATA_SET[(pin >> 5) & 1] = 1 << (pin & 0x1f);
}
```

Las funciones que acceden a los registros de control/estado de un dispositivo suelen ser muy cortas (1-2 líneas)

Interesa definirlas como inline para evitar la sobrecarga del paso de parámetros

Ejemplo de uso del driver del GPIO

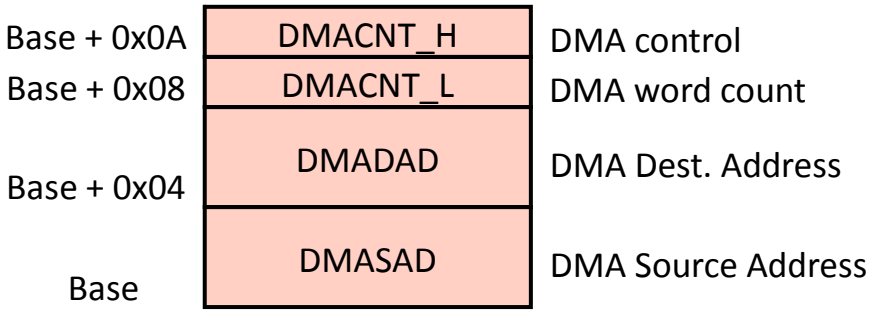
El hola mundo en la Redwire Econotag (ahora haciendo uso del driver del GPIO)

```
#include "system.h"

#define LED_RED       gpio_pin_44
#define LED_GREEN     gpio_pin_45

/* Función para esperar */
void esperar (void)
{
      int i, retardo = 100000;
      for (i=0 ; i<retardo ; i++);
}

int main ()
{
      /* Configuramos los pines de los leds para que sean de salida */
      gpio_set_pin_dir_output (LED_RED);
      gpio_set_pin_dir_output (LED_GREEN);

      while (1)
      {
            gpio_set_pin (LED_RED);
            gpio_clear_pin (LED_GREEN);

            esperar();

            gpio_set_pin (LED_GREEN);
            gpio_clear_pin (LED_RED);

            esperar();
      }

      return 0;
}
```

Contenidos

Estructuras

Adecuadas si hay una sola instancia del mismo dispositivo



Acceso a los registros

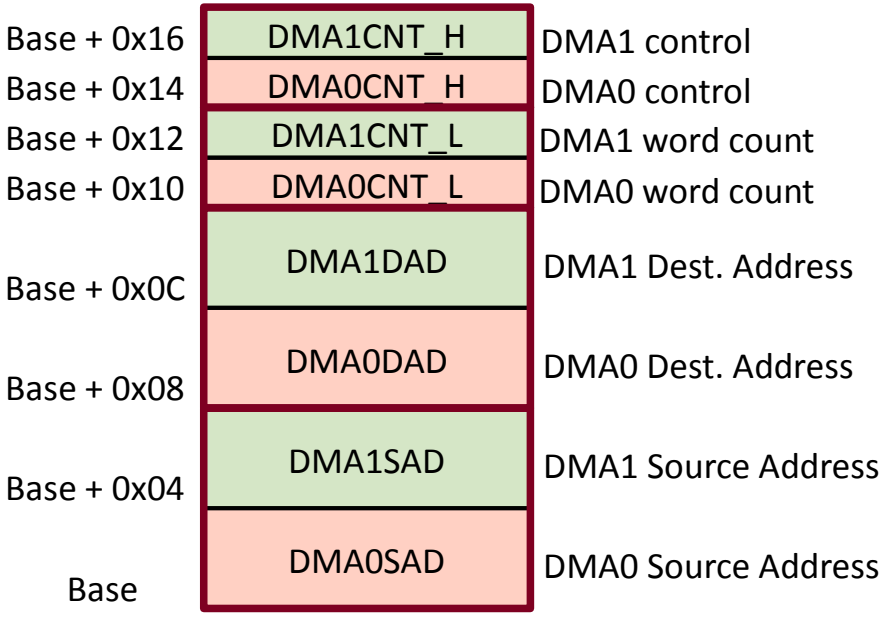
```
typedef struct
{
    uint32_t  DMA_SAD;
    uint32_t  DMA_DAD;
    uint16_t  DMACNT_L;
    uint16_t  DMACNT_H;
} dma_regs_t;

static volatile
dma_regs_t* const dma_regs = DMA_BASE;
```

Ejemplo de uso

```
inline void dma_memcpy(void* dest, void* source, uint32_t size)
{
  dma_regs->DMASAD   = (uint32_t) source;
  dma_regs->DMADAD   = (uint32_t) dest;
  dma_regs->DMACNT_L = size >> 2;
  dma_regs->DMACNT_H = DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32;
}
```

Estructuras de arrays

Adecuadas si hay varias instancias del mismo dispositivo y los registros del mismo tipo están agrupados



Acceso a los registros

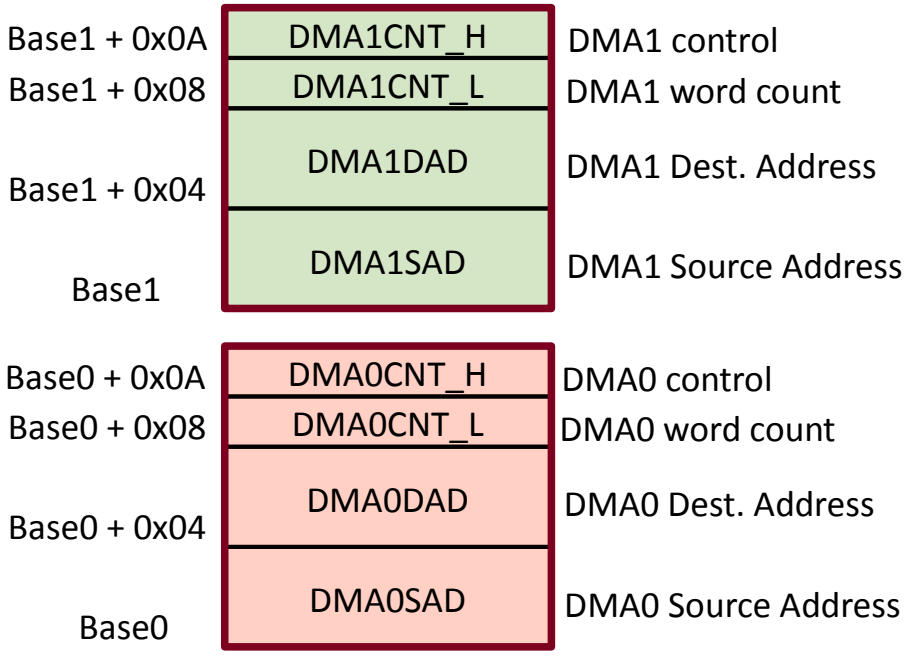
```
typedef struct
{
    uint32_t  DMA_SAD[2];
    uint32_t  DMA_DAD[2];
    uint16_t  DMACNT_L[2];
    uint16_t  DMACNT_H[2];
} dma_regs_t;

static volatile
dma_regs_t* const dma_regs = DMA_BASE;
```

Ejemplo de uso

```
inline void dma_memcpy(char ch, void* dest, void* source, uint32_t size)
{
  dma_regs->DMASAD[ch]   = (uint32_t) source;
  dma_regs->DMADAD[ch]   = (uint32_t) dest;
  dma_regs->DMACNT_L[ch] = size >> 2;
  dma_regs->DMACNT_H[ch] = DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32;
}
```

Arrays de punteros a estructuras

Adecuadas si hay varias instancias del mismo dispositivo y los registros de cada dispositivo están agrupados



Acceso a los registros

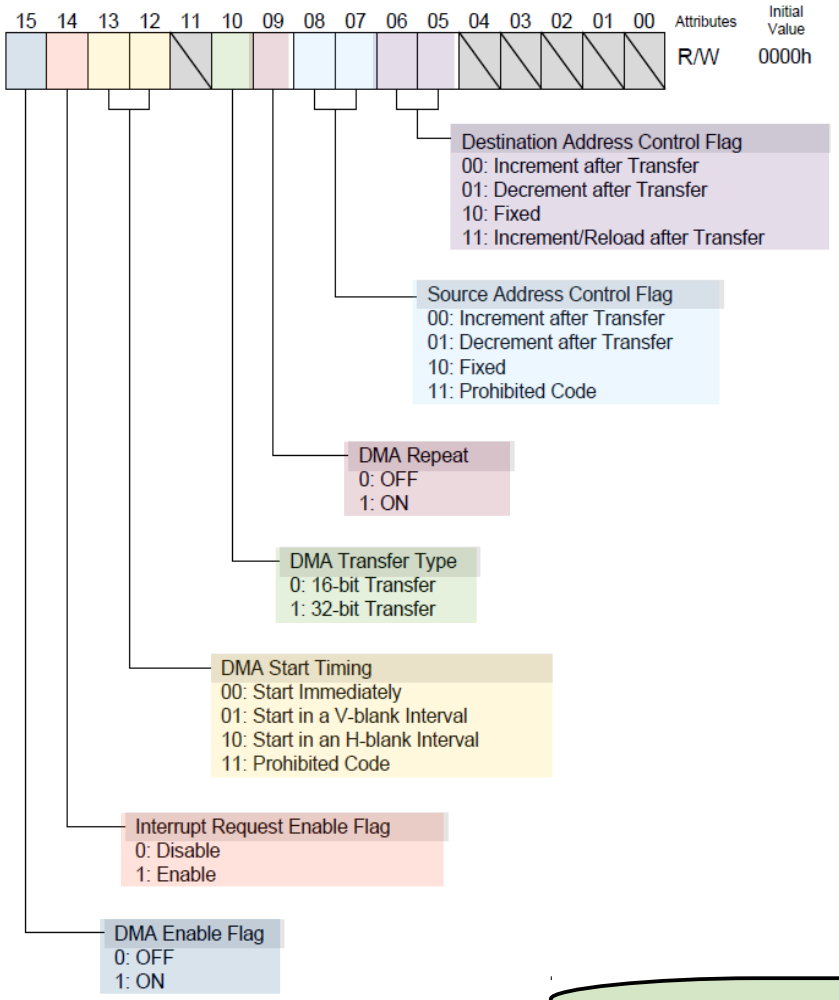
```
typedef struct
{
    uint32_t  DMA_SAD;
    uint32_t  DMA_DAD;
    uint16_t  DMACNT_L;
    uint16_t  DMACNT_H;
} dma_regs_t;

static volatile
dma_regs_t* const dma_regs[2] =
                {DMA0_BASE, DMA1_BASE};
```

Ejemplo de uso

```
inline void dma_memcpy(char ch, void* dest, void* source, uint32_t size)
{
  dma_regs[ch]->DMASAD   = (uint32_t) source;
  dma_regs[ch]->DMADAD   = (uint32_t) dest;
  dma_regs[ch]->DMACNT_L = size >> 2;
  dma_regs[ch]->DMACNT_H = DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32;
}
```

Acceso a campos dentro de un registro

DMAXCNT_H



Máscaras de bits

```
#define DMA_DEST_INCREMENT    ((uint16_t) (0 << 5))
#define DMA_DEST_DECREMENT    ((uint16_t) (1 << 5))
#define DMA_DEST_FIXED        ((uint16_t) (2 << 5))
#define DMA_DEST_RELOAD       ((uint16_t) (3 << 5))

#define DMA_SOURCE_INCREMENT  ((uint16_t) (0 << 7))
#define DMA_SOURCE_DECREMENT  ((uint16_t) (1 << 7))
#define DMA_SOURCE_FIXED      ((uint16_t) (2 << 7))

#define DMA_REPEAT            ((uint16_t) (1 << 9))

#define DMA_16                ((uint16_t) (0 << 10))
#define DMA_32                ((uint16_t) (1 << 10))

#define DMA_TIMING_IMMEDIATE  ((uint16_t) (0 << 12))
#define DMA_TIMING_VBLANK     ((uint16_t) (1 << 12))
#define DMA_TIMING_HBLANK     ((uint16_t) (2 << 12))

#define DMA_INT               ((uint16_t) (1 << 14))

#define DMA_ENABLE            ((uint16_t) (1 << 15))
```

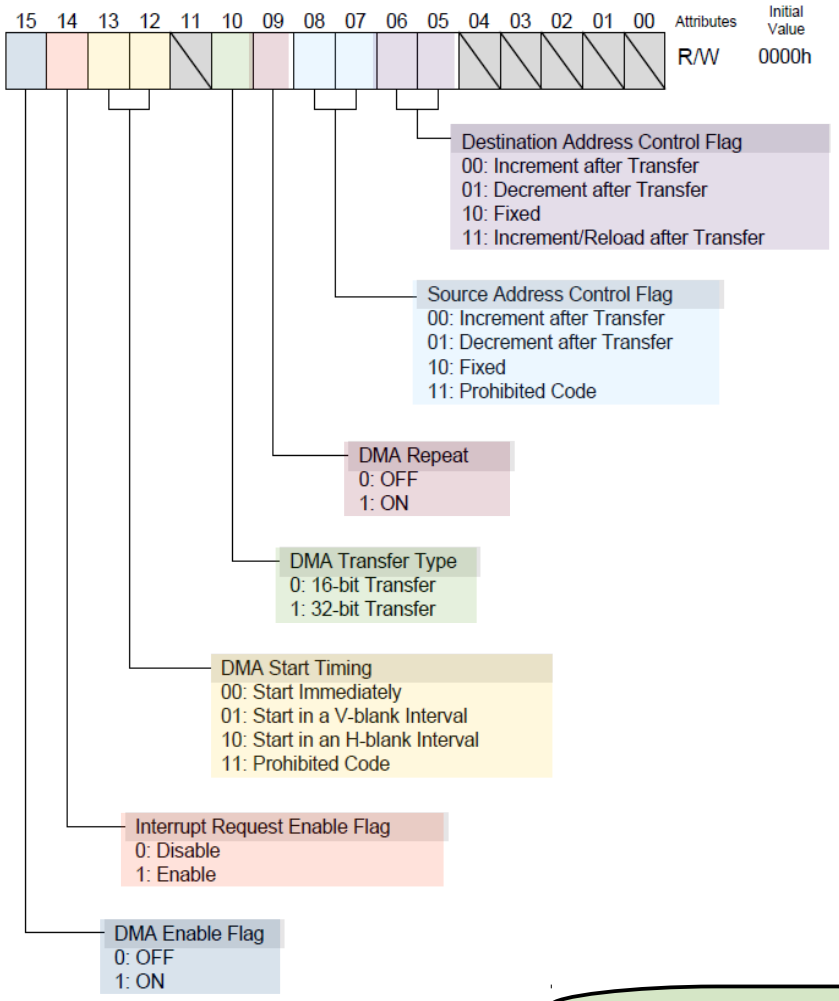
Permite modificar varios parámetros simultáneamente,
pero implica usar máscaras para leer un determinado atributo

Ejemplo de uso

```
dma_regs->DMACNT_H = DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32;
```

Acceso a campos dentro de un registro

DMAXCNT_H



Bitfields y estructuras anónimas

```
typedef struct
{
    uint32_t  DMA_SAD;
    uint32_t  DMA_DAD;
    uint16_t  DMACNT_L;
    struct
    {
        uint16_t          : 5;
        uint16_t DEST_CNT : 2;
        uint16_t SRC_CNT  : 2;
        uint16_t REPEAT   : 1;
        uint16_t TYPE     : 1;
        uint16_t          : 1;
        uint16_t TIMING   : 2;
        uint16_t IRQ_EN   : 1;
        uint16_t ENABLE   : 1;
    };
} dma_regs_t;

static volatile
dma_regs_t* const dma_regs = DMA_BASE;
```

Campos anónimos para padding

Estructura anónima

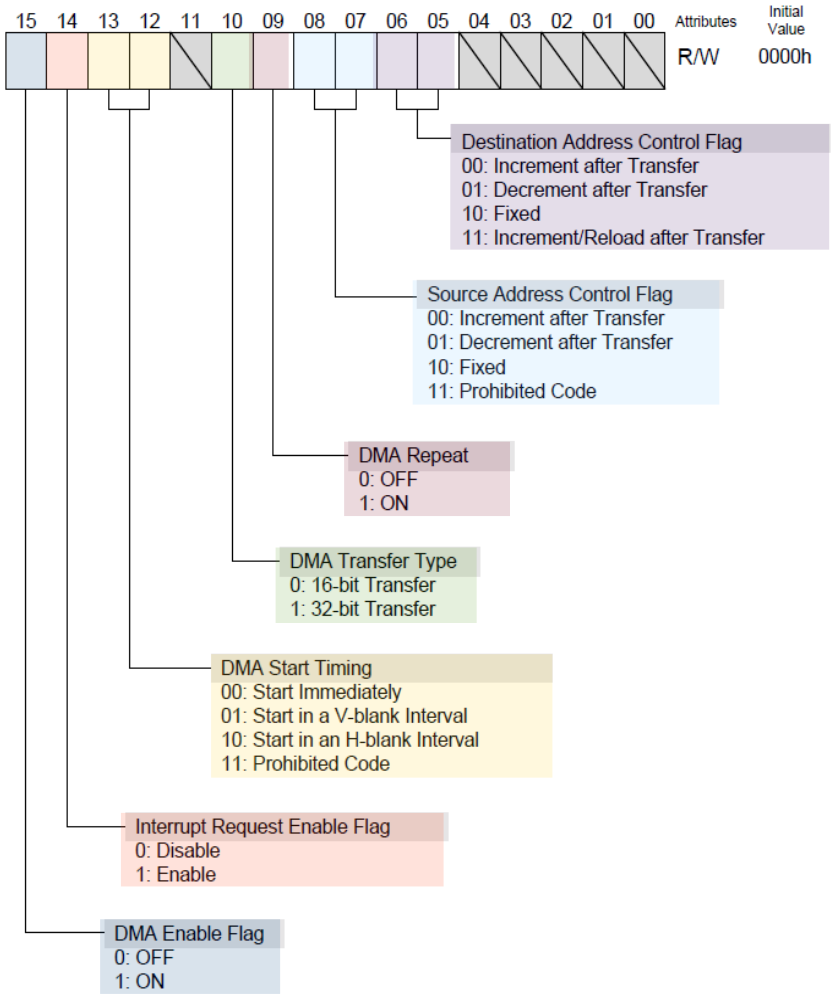
Evita el uso de máscaras de bits,
pero sólo permite acceder a un campo cada vez

Ejemplo de uso

```
dma_regs->TYPE = 0;    /* Se usará un bus de 16 bits para hacer la transferencia */
```

Acceso a campos dentro de un registro

DMAXCNT_H



Uniones anónimas

```
typedef struct
{
    uint32_t  DMA_SAD;
    uint32_t  DMA_DAD;
    uint16_t  DMACNT_L;
    union
    {
        struct
        {
            uint16_t          : 5;
            uint16_t DEST_CNT : 2;
            uint16_t SRC_CNT  : 2;
            uint16_t REPEAT   : 1;
            uint16_t TYPE     : 1;
            uint16_t          : 1;
            uint16_t TIMING   : 2;
            uint16_t IRQ_EN   : 1;
            uint16_t ENABLE   : 1;
        }
        uint16_t  DMACNT_H;
    };
} dma_regs_t;

static volatile
dma_regs_t* const dma_regs = DMA_BASE;
```

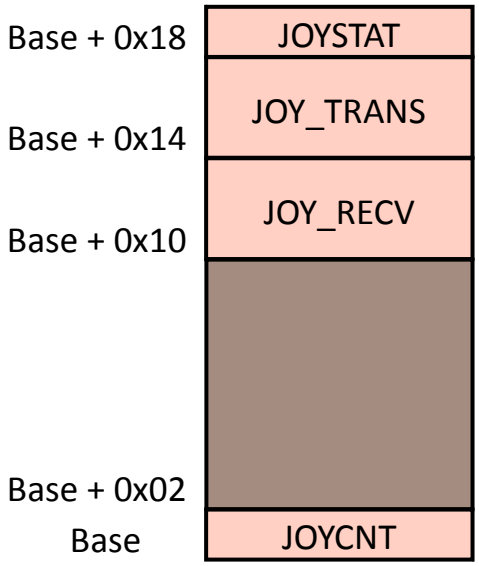
Estructura anónima

Union anónima

Ejemplo de uso

```
uint16_t t = dma_regs->TYPE;
dma_regs->DMACNT_H = DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32;
```

Padding



Más sencillo, aunque se crea un campo en la estructura que no se debe usar

Más seguro, aunque más complejo

Mediante arrays

```
typedef struct
{
    uint16_t  JOYCNT;
    uint16_t  padding[7];
    uint32_t  JOY_RECV;
    uint32_t  JOY_TRANS;
    uint16_t  JOYSTAT;
} joy_regs_t;

static volatile
joy_regs_t* const dma_regs = JOY_BASE;
```

Mediante bitfields anónimos

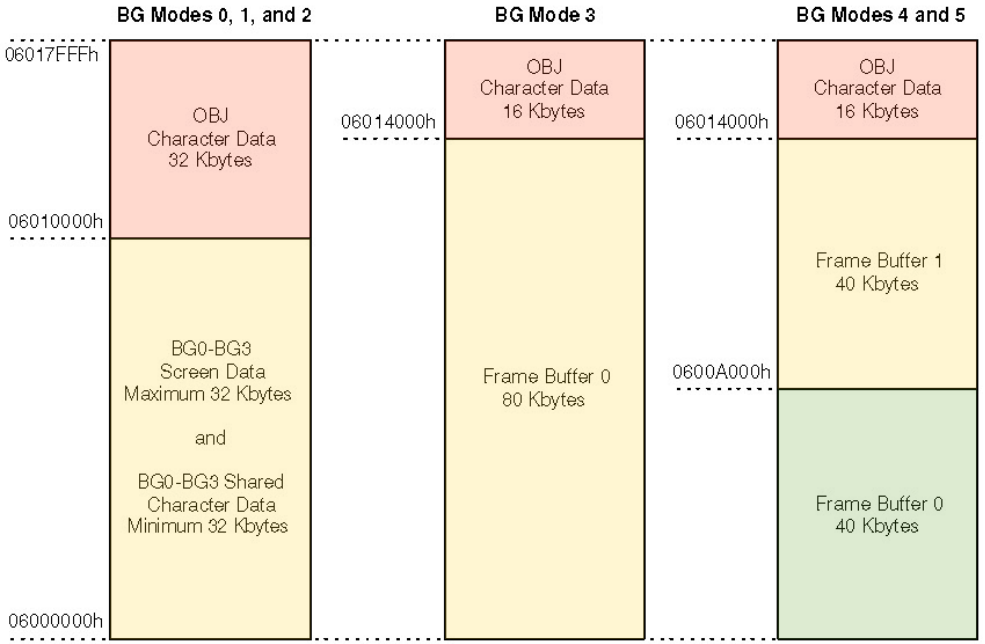
```
typedef struct
{
    uint16_t  JOYCNT;
    uint32_t  :1;
    uint32_t  :0;
    uint32_t  :1;
    uint32_t  :0;
    uint32_t  :1;
    uint32_t  :0;
    uint16_t  :1;
    uint32_t  JOY_RECV;
    uint32_t  JOY_TRANS;
    uint16_t  JOYSTAT;
} joy_regs_t;

static volatile
joy_regs_t* const dma_regs = JOY_BASE;
```

Reservamos un bit

Alineamos a la siguiente frontera

Gestión de búferes de memoria

Búfer de vídeo de la Gameboy Advance



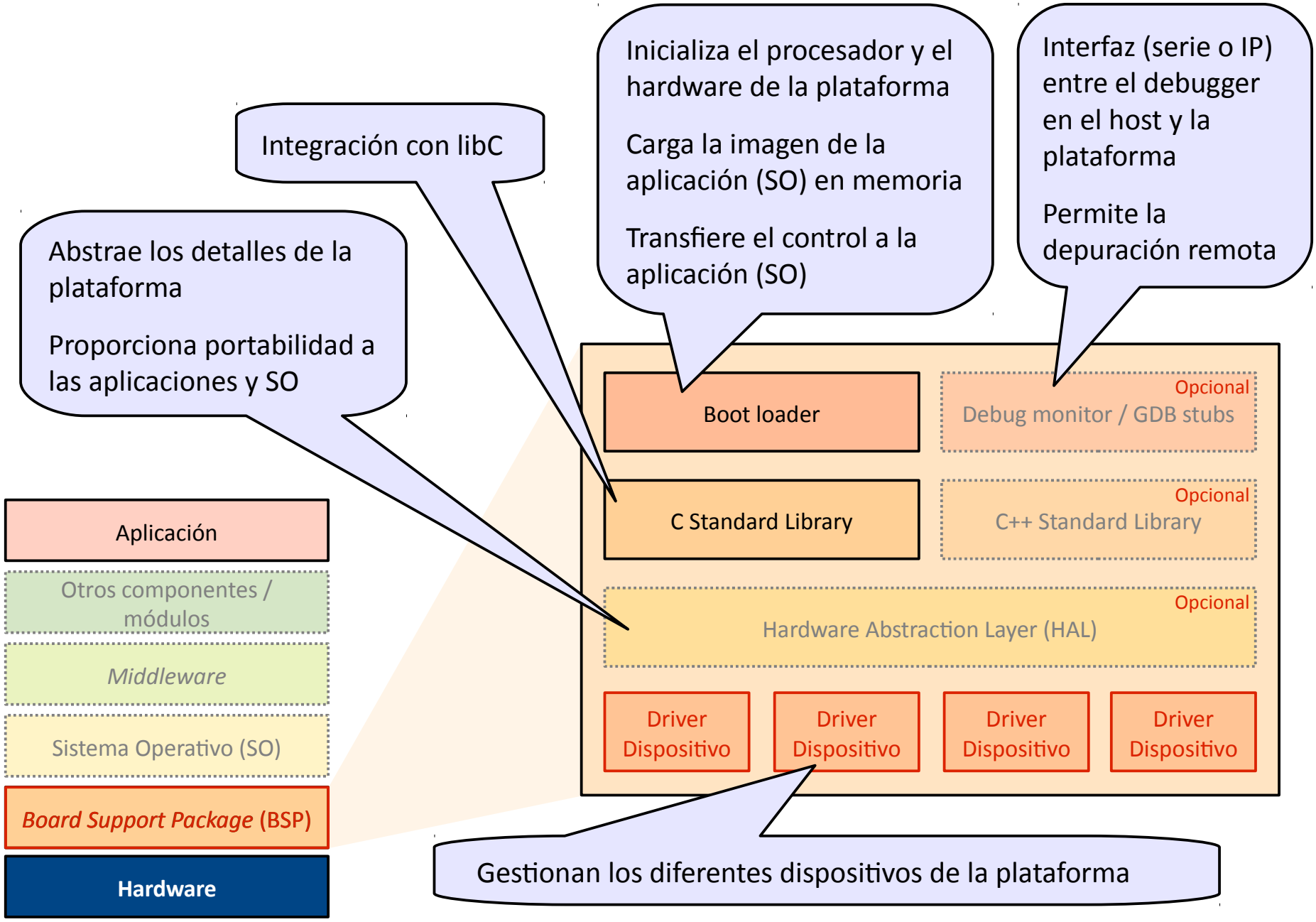
Fuente:
Nintendo of America Inc.
AGB Programming Manual, version 1.22, 1999 – 2001

Diferentes usos del búfer en función del modo de ejecución

```
typedef union {
  uint16_t video_buffer[BG_MODE_WIDTH3 * BG_MODE_HEIGHT3]; /* Modo 3 */

  struct { /* Modos 4 y 5 */
    uint16_t front_buffer[BG_MODE_WIDTH5 * BG_MODE_HEIGHT5];
    uint16_t back_buffer[BG_MODE_WIDTH5 * BG_MODE_HEIGHT5];
  };
} gba_vram_t;

static volatile gba_vram_t* const gba_vram = VRAM_BASE;
```

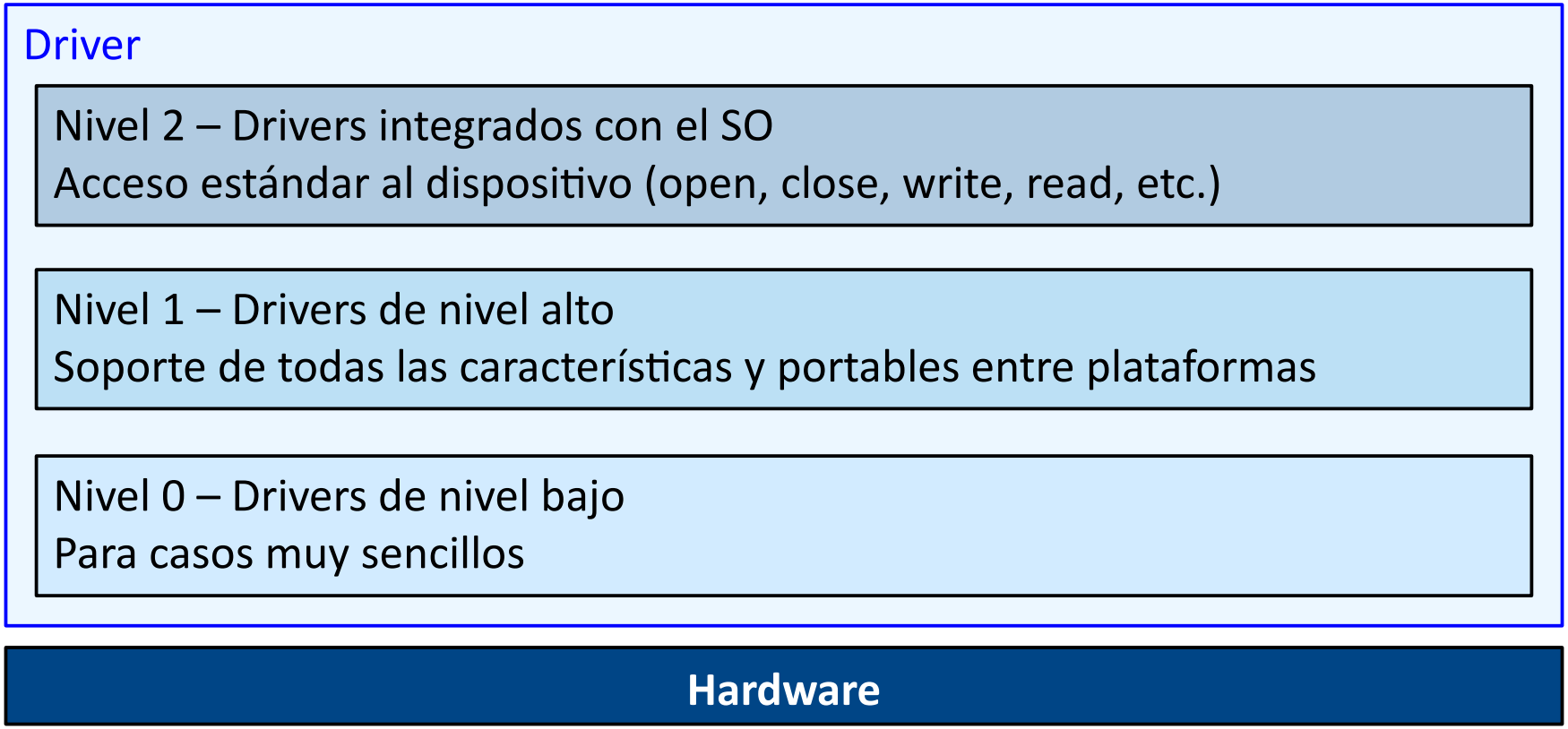
Contenidos

Componentes del *Board Support Package* (BSP)



Diseño de drivers en capas

El diseño en capas permite obtener desde drivers sencillos con un consumo mínimo de memoria (poco código) hasta drivers con soporte para todas las características del dispositivo, con un tamaño mayor del código



Máxima flexibilidad: se implementarán las capas necesarias en función de las características de la aplicación y las restricciones de la plataforma

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);
```

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)

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

Drivers de nivel 2

Acceso al dispositivo mediante llamadas a la biblioteca estándar de C (libC), proporcionando portabilidad y facilidad de uso

Características:

- Todos los dispositivos se tratan con un API estándar (ANSI C), independiente de la plataforma
- El uso de streams (FILE *) mejora las prestaciones de E/S
- E/S con formato (fprintf, fscanf)

Ejemplo

```
void function (void)
{
    FILE * uart_stream;
    uart_stream = fopen("/dev/uart1/9600,n,8,1", "w");
    fprintf(uart_stream, "Uso de la uart %d mediante streams de E/S\r\n", 1);
    fclose(uart_stream);
}
```

Lecturas recomendadas

M. Barr. *Portable Fixed-Width Integers in C*. Barr Group, 2007.
http://www.barrgroup.com/Embedded-Systems/How-To/C-Fixed-Width-Integers-C99/

D. Saks. *Mapping memory*. Embedded.com, 2004. http://www.embedded.com/electronics-blogs/programming-pointers/4025002/Mapping-memory/

D. Saks. *Mapping memory efficiently*. Embedded.com, 2004.
http://www.embedded.com/electronics-blogs/programming-pointers/4025027/Mapping-memory-efficiently/

D. Saks. *More ways to map memory*. Embedded.com, 2004.
http://www.embedded.com/electronics-blogs/programming-pointers/4025053/More-ways-to-map-memory/

D. Saks. *Padding and rearranging structure members*. Embedded.com, 2009.
http://www.embedded.com/design/prototyping-and-development/4008281/1/Padding-and-rearranging-structure-members/

N. Jones. *How to Use C's volatile Keyword*. Barr Group, 2007.
http://www.barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword/

M. Barr. *Combining C's volatile and const Keywords*. EmbeddedGurus, 2012.
http://embeddedgurus.com/barr-code/2012/01/combining-cs-volatile-and-const-keywords/

M. Barr. *How to Combine Volatile with Struct*. EmbeddedGurus, 2012.
http://embeddedgurus.com/barr-code/2012/11/how-to-combine-volatile-with-struct/

M. Barr. *Embedded C Coding Standard*. CreateSpace Independent Publishing Platform, 2008.