Entradas y salidas digitales

Sistemas con Microprocesadores

Mg.Ing. Esteban Volentini (evolentini@herrera.unt.edu.ar)

http://www.microprocesadores.unt.edu.ar/procesadores

Cronograma

Actividad	Inicio	Descripción	Completa
Presentación	24/08	Presentación y reglamento	✔
Tema 1	24/08	Estructura de las computadoras	✔
Tema 2	26/08	Proyecto con un microcontrolador	✔
Tema 3	31/08	Descripción funcional del ISA RV32IM	✔
Tema 4	07/09	Programación en lenguaje ensamblador	✔
Tema 5	18/09	Diseño del camino de datos de un procesador	✔
Tema 6	30/09	Diseño de la unidad de control del procesador	✔
Tema 7	05/10	Tecnologias y sistema de memoria	✔
Tema 8	09/10	Dispositivos de entrada/salida	✔
Tema 9	19/10	Entradas/salidas digitales (GPIO)	←
Tema 10	28/10	Generalidades de las interrupciones	
Tema 11	04/11	Interrupciones y excepciones en RISC-V	
Tema 12	13/10	Manejo de señales con temporizadores	
Tema 13	18/11	Transferencias de datos	
Proyectos	30/11	Seminarios de proyectos	

Temas a tratar

- Presentación del GD32VF103
- Descripción de las funciones GPIO
  - Registros de configuración de GPIO
  - Secuencia de configuración
    - En lenguaje ensamblador
  - Representación en lenguaje C
    - Usando macros, punteros, vectores y estructuras
- Bibliotecas HAL del fabricante

Arquitectura de una computadora



- La memoria almacena instrucciones y datos
- El procesador ejecuta las instrucciones para transformar los datos
- La entrada/salida permite la comunicación con el mundo exterior

Un poco de historia

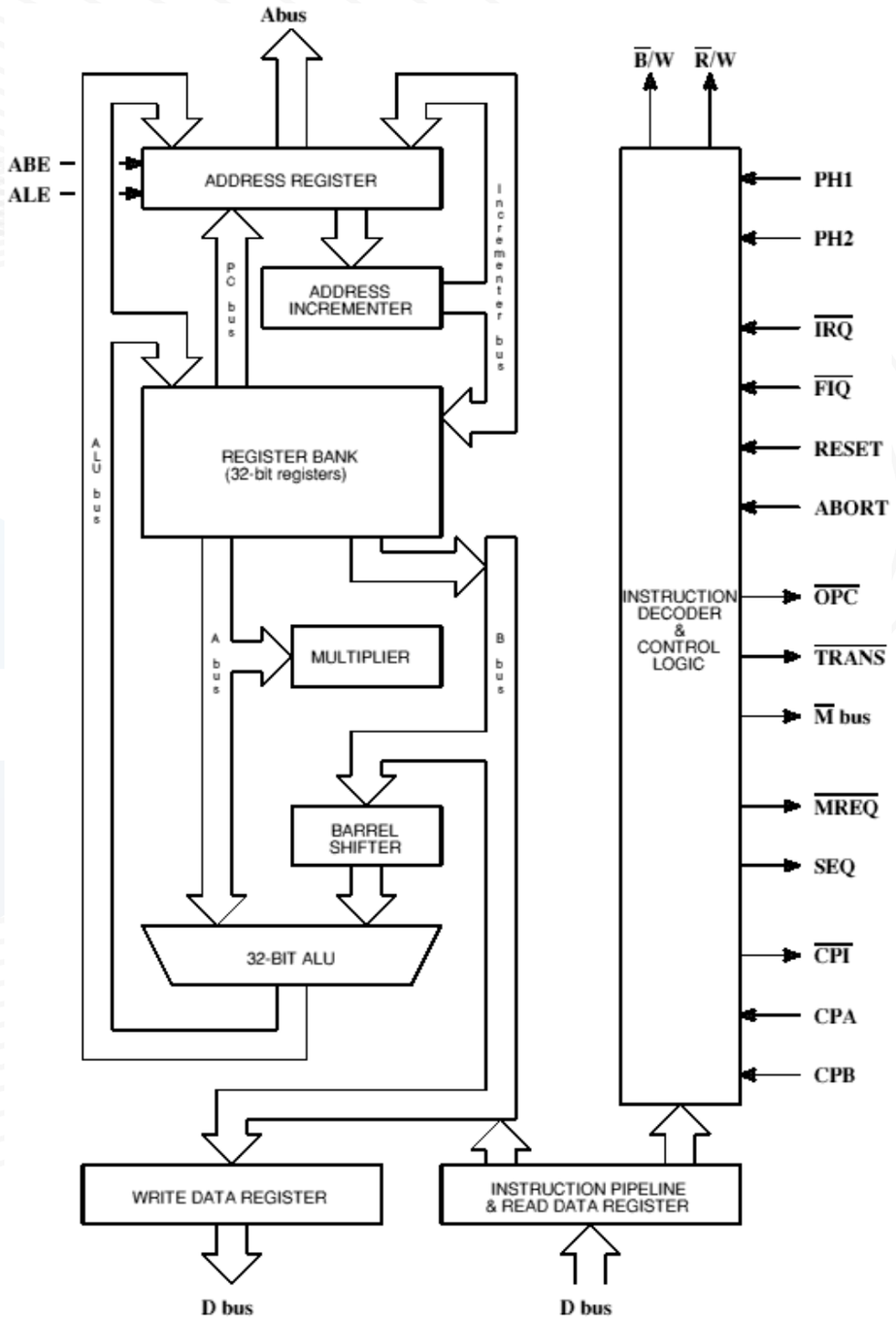
- Como se mostró en temas anteriores el procesador está formado por el camino de datos y el control
- En las primeras computadoras el procesador se implementaba utilizando componentes discretos
  - Un procesador estaba formado por muchos chips
- Esto limitaba la velocidad de los procesadores
  - Y aumentaba significativamente el consumo de energía

Un poco de historia

- El avance en la capacidad de miniaturización permitió aumentar la cantidad de componentes internos de un chip
- De esta forma en 1971 Intel produce el 4004, el primer microprocesador comercial de la historia
  - Un CPU de 4 bits de datos y 12 bits de direcciones
  - Operaba a una frecuencia de reloj de 70 KHz
  - Proceso de 10 um y un encapsulado de 16 terminales
- Tres años después, en 1974, se lanza el 8080
  - Un CPU de 8 bits de datos y 16 bits de direcciones
  - Operaba a una frecuencia de reloj de 2 MHz
  - Proceso de 6 um y un encapsulado de 40 terminales
  - Ancestro directo del Z80 y del 8086, el primer procesador de la IBM PC

Microprocesador (MPU)

- Integra en un solo componente:
  - Registros
  - Unidad aritmético-lógica
  - Sistema de control
- Es un procesador en un solo chip
- Require chips adicionales para
  - Memoria RAM y/o FLASH
  - Interfaces de Entrada/Salida
- Se comunica usando los buses de datos, direcciones y control
  - Se utiliza, principalmente, en computadoras de propósito general

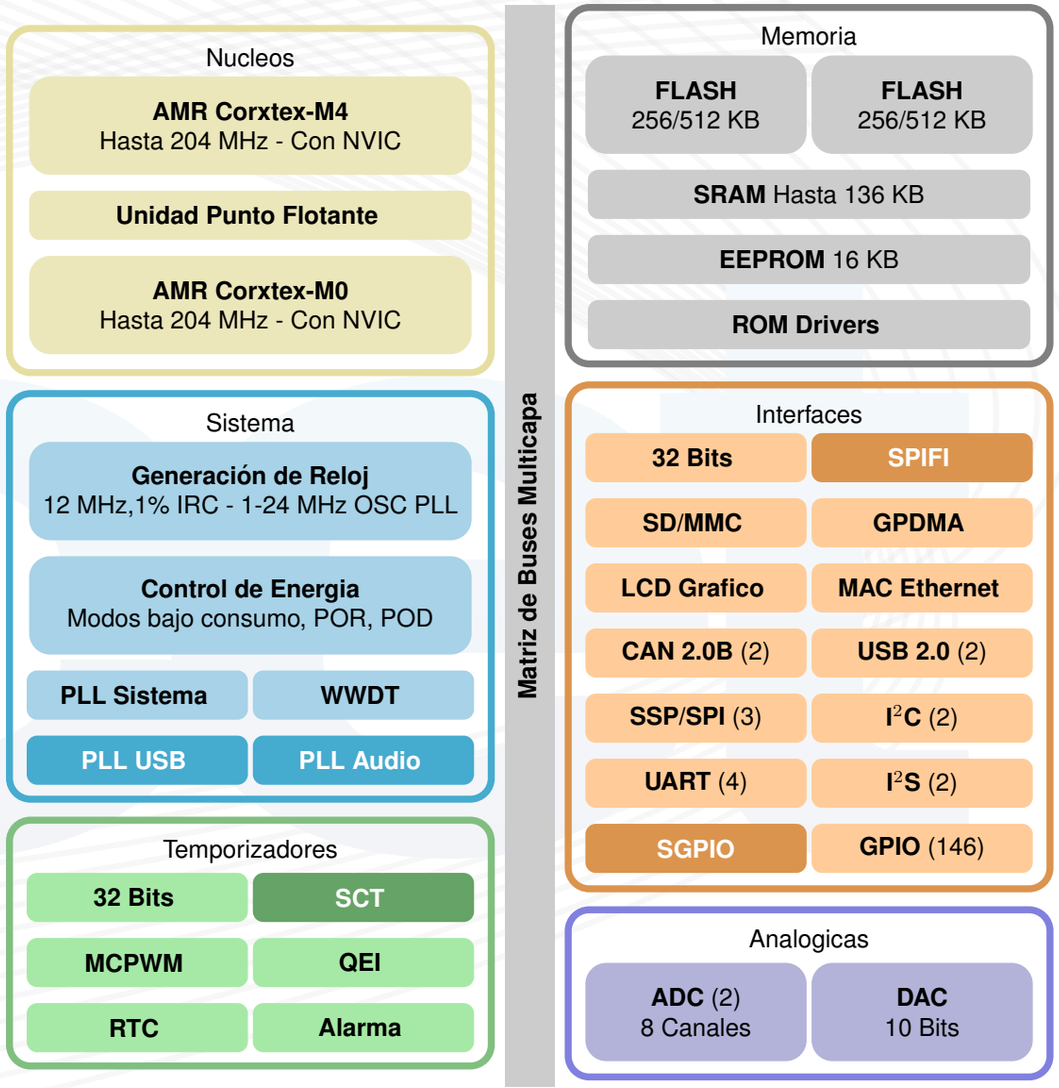


Un poco de historia

- La continua mejora en la capacidad de miniaturización permitió un nuevo avance:
- En vez de producir procesador mas potentes y complejos se podían integrar mas componentes y generar una solución mas simple de usar
- De esta forma en 1974 Texas Instruments lanza el TMS1000 que combina un procesador de 4 bits, 1 KByte de ROM, 32 Bytes de RAM, 23 líneas de GPIO y un generador de reloj
- En 1977 Intel presenta la familia MCS-48 y en 1980 la MCS51
  - El 8048, fue utilizado en millones de teclados de las PC
  - El 8051, fue clonado por mas de 20 fabricantes y todavía se utiliza en varias universidades para asignaturas equivalentes a la nuestra

Microcontrolador (MCU)

- Integra en un solo componente:
  - Microprocesador
  - Memoria RAM y ROM
  - Interfaces de Entrada/Salida
- Es una computadora en un solo chip
- Se comunica usando terminales especializados en diferentes funciones
  - Esta diseñado para ser usado en sistemas embebidos

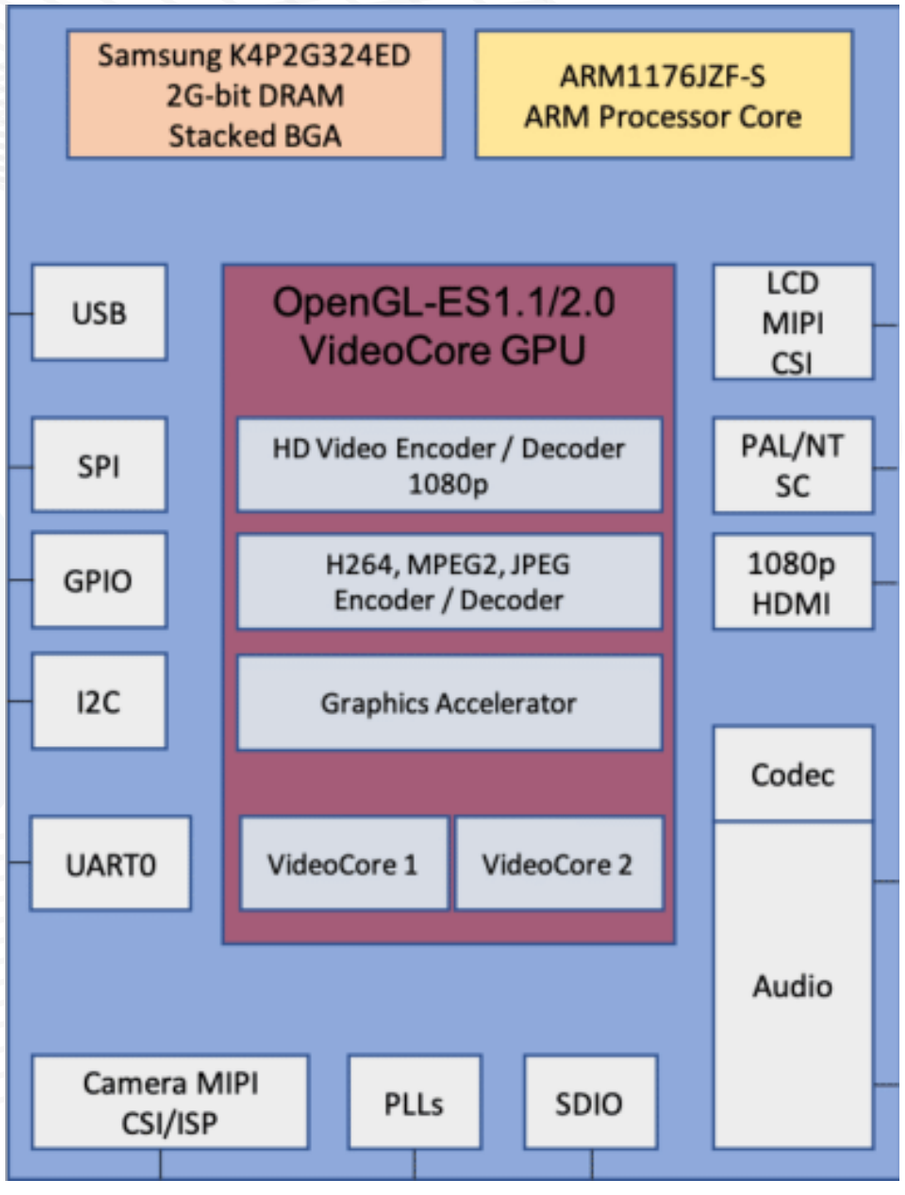


El panorama actual

- El aumento en la capacidad de integración continua
- Esto impulsa un aumento en el poder de calculo y una reducción en el consumo de los procesadores
  - En general los microcontroladores se benefician utilizando procesos de fabricación amortizados por los productos mas avanzados y caros
- Pero este aumento en la capacidad de integración abre una tercera opción
  - Integrar tambien todo lo necesario para resolver un problema concreto recurrente

System-On-Chip (SoC)

- Integra en un solo componente:
  - Microcontrolador
  - Dispositivos periféricos complejos como GPS, modems de comunicaciones o placas de video
- Es una solución completa y compleja en un chip
- Se utiliza en sistemas embebidos y de propósito general

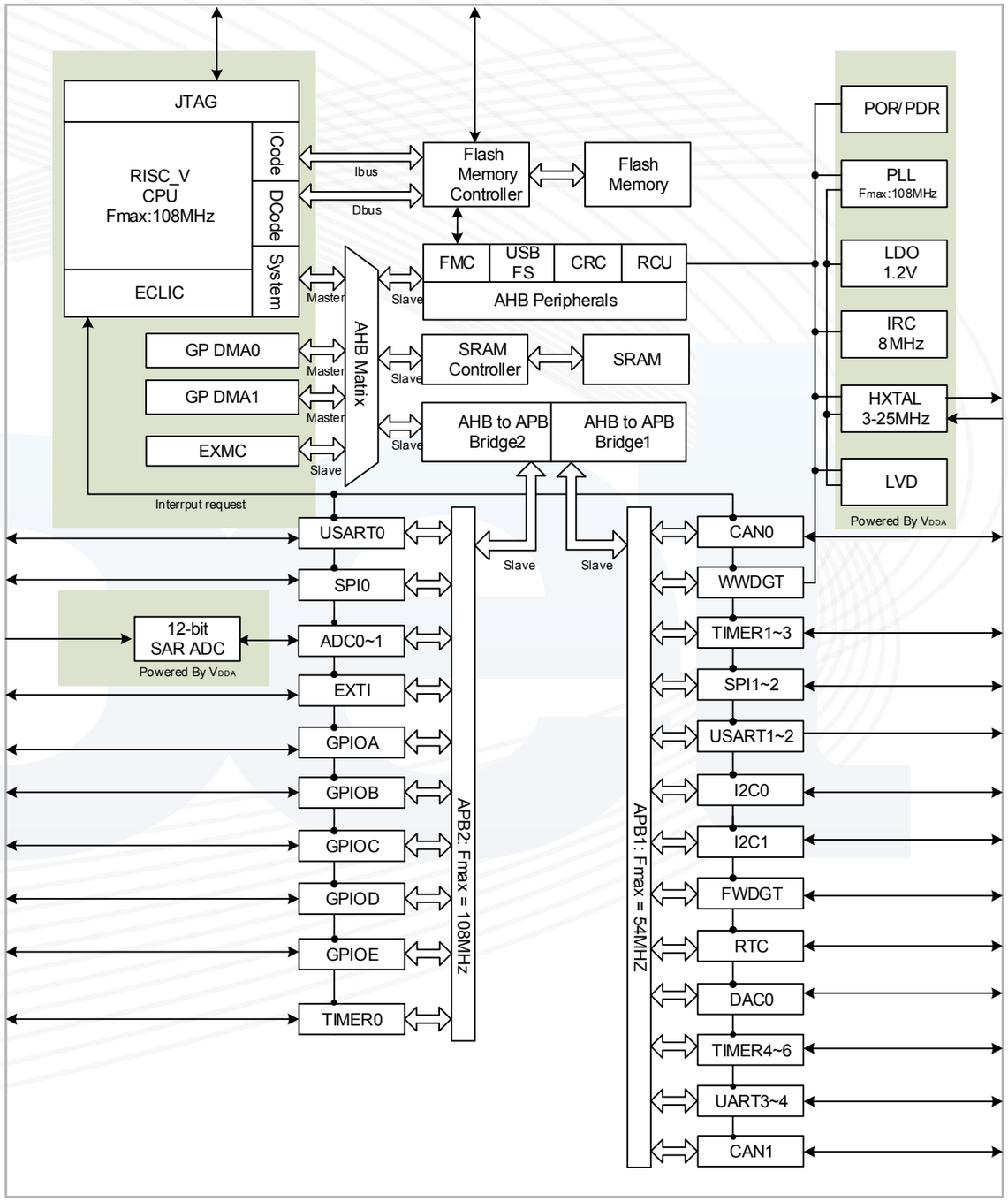


Microcontrolador GD32VF103

- En realidad se trata de una familia con 14 opciones diferentes:
  - 4 encapsulados: 36, 48, 64 y 100 terminales
  - 4 tamaños de FLASH:16, 32, 64 y 128 KiB
  - 4 tamaños de RAM:6, 10, 20 y 32 KiB
- Utiliza un procesador RISC-V de 32 bits denominado N200 Bumblebee
  - Desarrollado por Nuclei System Technology y Andes Technology
  - Licenciado por Gigadevices Semiconductors Inc
- Es una variante de la familia GD32F1xx a GD32F4xx
  - Clones de los STM32F1xx a STM32F4xx de ST Semiconductors

Microcontrolador GD32VF103

- Hasta 80 puertos GPIO
- Hasta 16 entradas analógicas de 12 bits
- Hasta 6 temporizadores para generar y analizar señales
- Hasta 2 puertos CAN
- Hasta 5 puertos USART
- Hasta 2 puertos I2C
- Hasta 1 puerto SPI
- Un puerto USB OTG



Entradas y salidas digitales

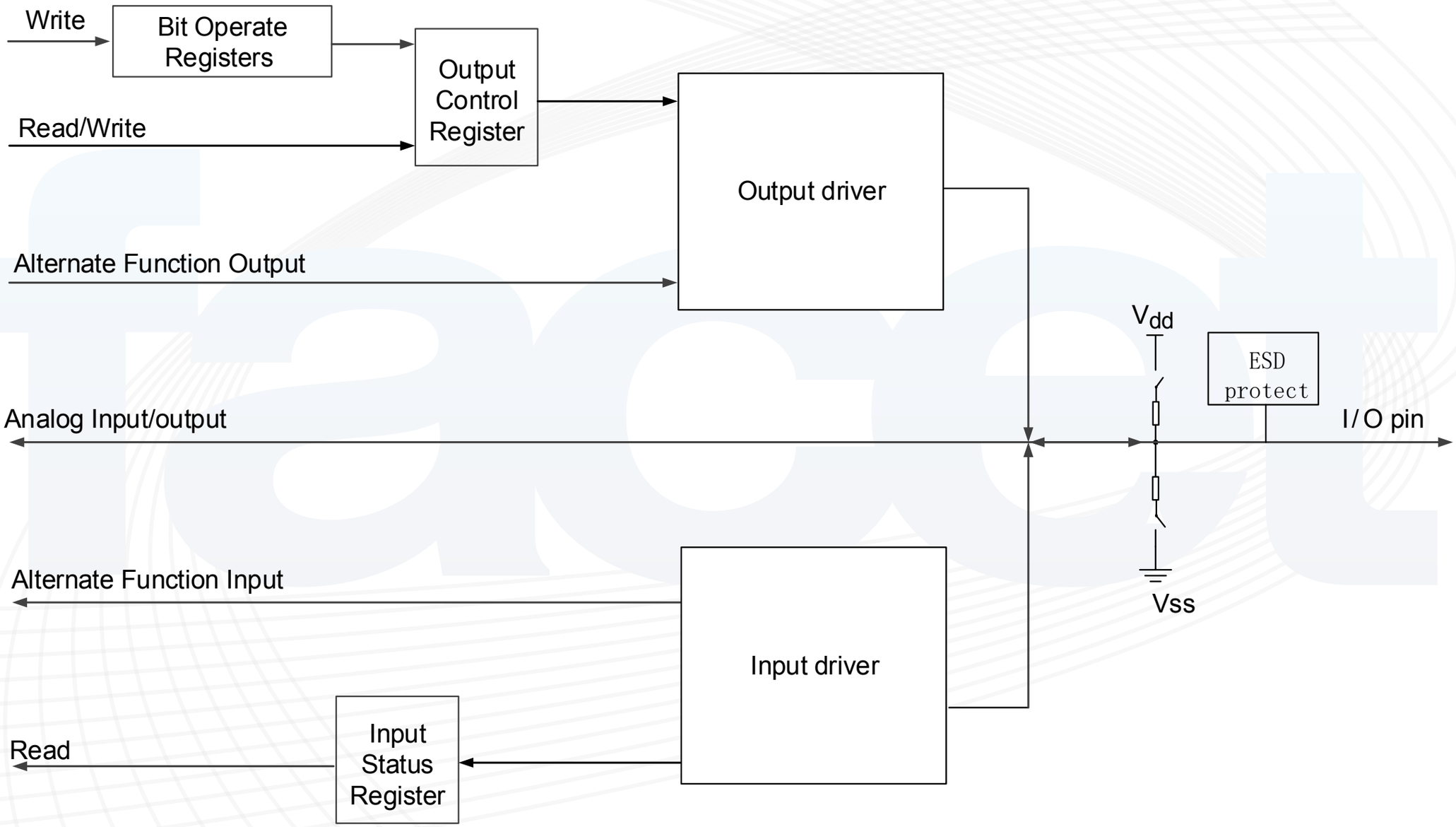
- Todos los microcontroladores integran puertos con entradas y salidas digitales
  - Conocidos como GPIO (General Purpose Input/Output)
  - Similares a los que se estudiaron en el tema anterior
- En general se puede configurar cada linea individual del puerto como entrada o como salida
- Es habitual que se incluyan funciones adicionales:
  - Resistencias de *pull-up* y de *pull-down* en las entradas
  - Limitación en la corriente y el *slew rate* de las salidas
- Se necesita la información del dispositivo
  - En dos documentos: la hoja de datos y el manual de usuario

Buscando más información

- En la hoja de datos se explica que este dispositivo puede tener hasta 80 terminales GPIO llamados de PA0 hasta PE15
  - Tiene 5 puertos, de PA a PE, y cada puerto tiene 16 lineas, de Px0 a Px15
- La información sobre el manejo de las entradas y salidas digitales se encuentra en el capitulo 7 del manual de usuario
  - Al inicio explica que cada terminal puede operar como entrada/salida digital, entrada analógica o entrada/salida con una función alterna
  - Después se explica como mover cada dispositivo a los diferentes terminales
- Una primera configuración es común para cualquier función y corresponde a las características eléctricas del terminal
  - En este tema esta configuración será mas simple porque en este procesador la función de GPIO ya está asignada por defecto
- La segunda configuración depende de la función asignada al terminal
  - En este tema vamos a analizar las entradas y salidas digitales (GPIO)

Estructura básica de los terminales

Figure 7-1. Basic structure of a general-pupose I/O



Buscando más información

- En la sección 7.5 se describen los registros de control cada puerto GPIO
  - Todos tiene la misma estructura
  - Cada puerto tiene asignada una dirección *base*
- Cada puerto tiene en total 7 registros
  - Todos los registros son de 32 bits y están en direcciones de memoria consecutivas a partir de la dirección *base*
  - Para cada registro se especifica un desplazamiento
  - La dirección efectiva de un registro se obtiene sumando el desplazamiento a la dirección base del puerto

Registros de control del GPIO

- **CTL**: controlan la configuración de cada terminal: si es entrada o salida, resistencias de entradas, configuración y velocidad de salidas
- **ISTAT**: permiten leer el estado actual real de todos los terminales del puerto
- **OCTL**: permite leer y escribir el estado actual de cada terminal de salida
- **BOP, BC**: permiten cambiar el estado de algunos terminales de salida sin alterar el resto
- **LOCK**: permite bloquear la configuración de los terminales hasta el siguiente reinicio del procesador

Registros de control del GPIO

Dirección	Nombre	Reset	Acceso
0x00	GPIOx_CTL0	0x4444.4444	Lectura y escritura
0x04	GPIOx_CTL1	0x4444.4444	Lectura y escritura
0x08	GPIOx_ISTAT	0x0000.????	Solo lectura
0x0C	GPIOx_OCTL	0x0000.0000	Lectura y escritura
0x10	GPIOx_BOP	0x0000.0000	Solo escritura
0x14	GPIOx_BC	0x0000.0000	Solo escritura
0x1C	GPIOx_LOCK	0x0000.0000	Lectura y Escritura

Configuración basica de los terminales

MODO DE USO		MD[1:0]	CTL[1:0]	OCTL
Entrada	Analógica	00	00	No importa
	Flotante	01		No importa
	Pull Down	10		0
	Pull Up	10		1
Salida de propósito general (GPIO)	Push Pull	00	01: Hasta 10 MHz 10: Hasta 2 MHz 11: Hasta 50 MHz	0 o 1
	Colector Abierto	01		0 o 1
Salida de función alterna	Push Pull	10		No importa
	Colector Abierto	11		No importa

GPIOx_CTLx

GPIOx_CTL0 (Offset 0x00)

31:30	29:28	...	7:6	5:4	3:2	1:0
CTL7	MD7	. . .	CTL1	MD1	CTL0	MD0

Terminal 7 | Terminal 1 | Terminal 0

GPIOx_CTL1 (Offset 0x04)

31:30	29:28	...	7:6	5:4	3:2	1:0
CTL15	MD15	. . .	CTL9	MD9	CTL8	MD8

Terminal 15 | Terminal 9 | Terminal 8

GPIOx_CTLx

GPIOx_CTLx

31:30	29:28	...	7:6	5:4	3:2	1:0
CTLz	MDz	. . .	CTLy	MDy	CTLx	MDx

Modo de funcionamiento	Valor
Modo entrada (por defecto)	**0x0**
Modo salida, frecuencia máxima 10 MHz	**0x1**
Modo salida, frecuencia máxima 2 MHz	**0x2**
Modo salida, frecuencia máxima 50 MHz	**0x3**

GPIOx_CTLx

GPIOx_CTLx

31:30	29:28	...	7:6	5:4	3:2	1:0
CTLz	MDz	. . .	CTLy	MDy	CTLx	MDx

Modo de funcionamiento	Valor	Modo
Entrada analógico	0x0	Entrada
Entrada flotante	0x1	Entrada
Entrada con pull-down o pull-up (Por defecto)	0x2	Entrada
Reservada	0x3	Entrada
Salida GPIO con push-pull	0x0	Salida
Salida GPIO a colector abierto	0x1	Salida
Salida de función alterna con push-pull	0x2	Salida
Salida de función alterna a colector abierto	0x3	Salida

GPIOx_ISTAT y GPIOx_OCTL

GPIOx_ISTAT (Offset 0x08)

31	...	16	15	...	1	0
Reservado			ISTAT15	. . .	ISTAT1	ISTAT0

Función	Valor
El nivel actual de la entrada es bajo	**0**
El nivel actual de la entrada es alto	**1**

GPIOx_OCTL (Offset 0x0C)

31	...	16	15	...	1	0
Reservado			OCTL15	. . .	OCTL1	OCLT0

Función	Valor
El nivel actual de la salida debe ser bajo	**0**
El nivel actual de la salida debe ser alto	**1**

GPIOx_BOP y GPIOx_BC

GPIOx_BOP (Offset 0x10)

31	...	17	16	15	...	1	0
CR15	. . .	CR1	CR0	BOP15	. . .	BOP1	BOP0

Función	Valor
No cambia el nivel de la salida	0
Fija el nivel de la salida en alto	1

Valor	Función
0	No cambia el nivel de la salida
1	Fija el nivel de la salida en bajo

GPIOx_BC (Offset 0x14)

31	...	16	15	...	1	0
Reservado			CR15	. . .	CR1	CR0

GPIOx_LOCK

GPIOx_LOCK (Offset 0x1C)

31	...	17	16	15	...	1	0
Reservado			LKK	LK15	. . .	LK1	LK0

Solo puede escribirse siguiendo una secuencia específica:

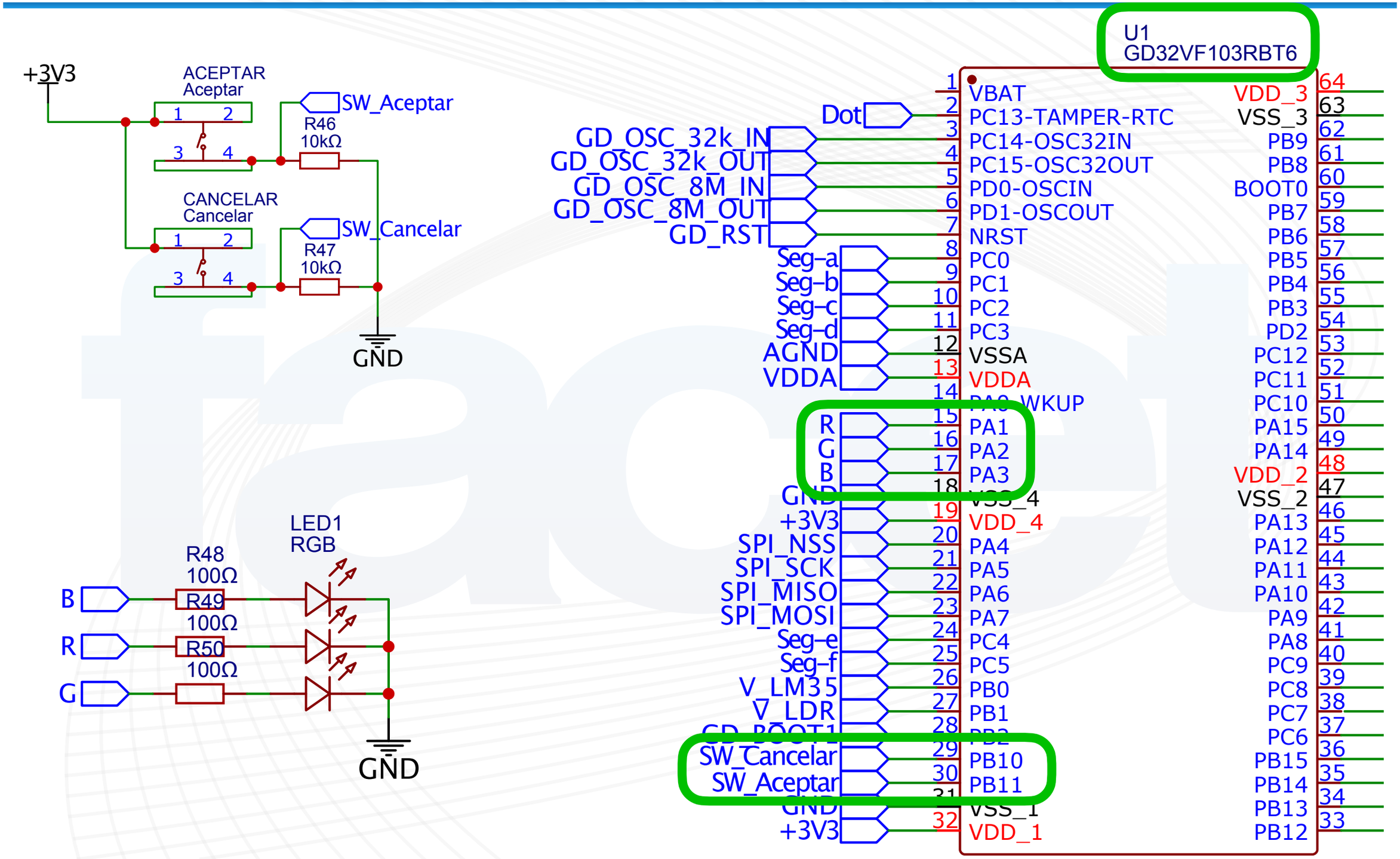
WR 1 → WR 0 → WR 1 → RD 0 → RD 1

Durante este proceso los bits LK[15:0] debe mantener su valor

Valor	Función
0	El registro no está bloqueado y la configuración de los terminales se puede cambiar
1	El registro está bloqueado y los terminales bloqueado mantiene la configuración hasta el reinicio

Función	Valor
La configuración del terminal no está bloqueada y se puede cambiar	**0**
La configuración del terminal está bloqueada hasta el reinicio	**1**

Un ejemplo concreto



Un ejemplo concreto

- A partir del diagrama esquemático de la placa sabemos que:
  - La tecla Aceptar está conectada al terminal 30 y que está etiquetado como PB11
  - El canal rojo del led RGB está conectado al terminal 15 y que está etiquetado como PA1
  - El microcontrolador es un GD32VF103RBT6 y que tiene una cápsula de 64 terminales
- Las etiquetas del esquemático no necesariamente coinciden con el nombre real del terminal, a veces se asignan por la función

Un ejemplo concreto

- Con la hoja de datos del dispositivo se puede saber que:
  - El dispositivo tiene 32 KiB de RAM y 128 KiB de Flash (Página 8)
  - Que el terminal 15 se llama PA1, que su función por defecto es como GPIO y que ademas puede ser una entrada analógica, una entrada o salida de un temporizador o una linea de control en un puerto de comunicaciones (Página 26)
  - Que el terminal 30 se llama PB11, que su función por defecto es como GPIO y que ademas puede ser una señal de datos de puerto de comunicaciones o una entrada o salida de un temporizador (Página 27)
- En este procesador los terminales se identifican usando la función de GPIO

Un ejemplo concreto

	Aceptar	Cancelar	LED R	LED G	LED B
Terminal	PB11	PB10	PA1	PA2	PA3
Funcion	GPIO	GPIO	GPIO	GPIO	GPIO
Modo	Entrada	Entrada	Salida	Salida	Salida
Velocidad	N/A	N/A	2MHz	2MHz	2MHz
Pull Up/Down	No	No	N/A	N/A	N/A
Tipo de Salida	N/A	N/A	Push-Pull	Push-Pull	Push-Pull
MDx	0x0	0x0	0x2	0x2	0x2
CTLx	0x0	0x0	0x0	0x0	0x0

La secuencia de configuración

- Para configurar la salida del led rojo (PA1)
  - Escribir el patrón 0x02 en los bits 4 a 7 del registro GPIOA_CLR0
- Para prender el led rojo (PA1)
  - Poner en 1 el bit 1 del registro GPIOA_OCTL
  - o escribir el valor 0x0000002 en el registro GPIOA_BOP
- Para apagar el led rojo (PA1)
  - Poner en 0 el bit 1 del registro GPIOA_OCTL
  - o escribir el valor 0x0020000 en el registro GPIOA_BOP
  - o escribir el valor 0x0000002 en el registro GPIOA_BC
- Para configurar la entrada de la tecla aceptar (PB11)
  - Escribir el patrón 0x00 en los bits 12 a 15 del registro GPIOB_CLR1
- Para leer el estado de la tecla tecla aceptar (PB11)
  - Analizar el bit 11 del registro GPIOA_ISTAT

La secuencia en ensamblador

- En lenguaje ensamblador es fácil leer y escribir en una dirección de memoria especifica
- Para facilitar la legibilidad conviene asignar nombres a las direcciones de memoria
- Gracias a la estructura regular de los dispositivos, y al modo de direccionamiento indexado, es mas conveniente definir dos nombres:
  - Uno para la dirección base del dispositivo
  - Otro para el desplazamiento del registro concreto
- De esta forma se apunta un solo registro a la base del dispositivo y se puede acceder a todos los registros usando el desplazamiento en cada operación de lectura o escritura

La secuencia en ensamblador

```
.equ   GPIOA_BASE,0x40010800 # Direccion base GPIOA
.equ   GPIOB_BASE,0x40010C00 # Direccion base GPIOB
.equ   GPIO_CTL0,0x00        # Control de terminales 0 a 7
.equ   GPIO_BOP,0x10         # Puesta a 1 de las salidas

configurar_led:
  la   t0,GPIOA_BASE    # Apunta a la base de GPIOA
  lw   t2,GPIO_CTL0(t0) # Carga el registro de control
  andi t2,t2,0xFFFFFF0F # Borra los 4 bits del pin 1
  andi t2,t2,0x00000020 # Configura los 4 bits del pin 1
  sw   t2,GPIO_CTL0(t0) # Actualiza registro de control
prender_led:
  la   t0,GPIOA_BASE    # Apunta a la base de GPIOA
  li   t2,0x00000002    # Valor para prender el LED Rojo
  sw   t2,GPIO_BOP(t0)  # Escribe el valor en las salidas
```

Probando en la placa

- El entorno de desarrollo de la placa está configurado para compilar un proyecto escrito en lenguaje C
  - Cuando arranca el procesador se ejecuta la función **`main.c`**
  - ¡¡¡Esta función nunca debe terminar!!!
- Desde una función escrita en C se puede llamar a una función escrita en ensamblador si se respeta el ABI IPL32
- Para que la rutina de configuración sea reutilizable no debería alterar ningún bit de ningún puerto que no sean los que se quieren configurar
  - Debería recibir como argumento sobre que terminal debe operar
- ¡¡¡Cuidado!!! Para controlar el consumo el procesador inicia con la señal de reloj apagada en los dispositivos de entrada/salida

Acceso a los dispositivos en C

- En lenguaje C un puntero permite leer o escribir el contenido de una dirección de memoria arbitraria
- Cualquier entero de 32 bits se puede considerar una dirección de memoria
- Un *cast* es una facilidad del leguaje C que permite convertir un tipo de datos en otro diferente
- Se podría acceder a los registros haciendo un cast de una constante con la dirección de memoria del registro para convertirla en un puntero a un entero de 32 bits
  - Esta constante podría ser tambien la suma de dos constantes
- El lenguaje C tambien define operadores lógicos a nivel de bits
  - Esto permite utilizar mascaras de bits igual que en lenguaje ensamblador

La secuencia en lenguaje C

```c
#define GPIOA_BASE 0x40010800  // Direccion base GPIOA
#define GPIOB_BASE 0x40010C00  // Direccion base GPIOA
#define GPIO_CTL0  0x00        // Control de terminales 0 a 7
#define GPIO_BOP   0x10        // Puesta a 1 de las salidas

void configurar_led(int pin) {
    unsigned int valor;
    valor = *(unsigned int *)(GPIOA_BASE + GPIO_CTL0);
    valor &= ~(0xF << (pin << 2));
    valor |= (0x2 << (pin << 2));
    *(unsigned int *)(GPIOA_BASE + GPIO_CTL0) = valor;
}

void prender_led(int pin) {
    *(unsigned int *)(GPIOA_BASE + GPIO_BOP) = (1 << pin);
}
```

Esto no siempre va a funcionar

- En la declaración anterior el compilador de C asume que se está accediendo a variables en memoria
- Pero los puertos de entrada/salida tienen una diferencia muy importante
  - Pueden cambiar su valor en forma espontánea
- Este comportamiento no es compatible con las optimizaciones del compilador
  - Por ejemplo al leer la misma variable dentro de un lazo que no asigna valor el compilador asume que la variable nunca cambia
- Por eso es necesario agregar la directiva **`volatile`** que avisa al compilador de este comportamiento en la variable

La secuencia en lenguaje C

```c
#define GPIOA_BASE 0x40010800  // Direccion base GPIOA
#define GPIOB_BASE 0x40010C00  // Direccion base GPIOA
#define GPIO_CTL0  0x00        // Control de terminales 0 a 7
#define GPIO_BOP   0x10        // Puesta a 1 de las salidas

void configurar_led(int pin) {
    unsigned int valor;
    valor = *(volatile unsigned int *)(GPIOA_BASE + GPIO_CTL0);
    valor &= ~(0xF << (pin << 2));
    valor |= (0x2 << (pin << 2));
    *(volatile unsigned int *)(GPIOA_BASE + GPIO_CTL0) = valor;
}

void prender_led(int pin) {
    *(volatile unsigned int *)(GPIOA_BASE + GPIO_BOP) = (1 << pin);
}
```

Mejorando la legibilidad

- El código anterior es bastante críptico
  - Sería conveniente mejorar la legibilidad
- El compilador de C nos brinda algunas facilidades para esto
  - Los macros ademas de sustituir valores numéricos por constantes con nombres puede recibir argumentos
  - Esto permite definir pseudo-funciones que reciben argumentos

La secuencia usando macros

```
#define GPIOA_BASE       0x40010800 // Direccion base GPIOA
#define GPIO_CTL0        0x00       // Control de terminales 0 a 7
#define GPIO_BOP         0x10       // Puesta a 1 de las salidas
#define PUSH_PULL_2MHZ   0x02       // Salida push-pull de 2 MHz

#define GPIO_CTL_SET(value, pin)  (value << ((pin & 7) << 2))
#define GPIO_CTL_MASK(pin)        ~GPIO_CTL_SET(0x0F, pin)
#define GPIO_BIT_MASK(pin)        (1 << pin)
```

```
#define GPIOA(x)  *(volatile unsigned int *)(GPIOA_BASE + x)

void configurar_led(int pin) {
    unsigned int valor = GPIOA(GPIO_CTL0);
    valor &= GPIO_CTL_MASK(terminal);
    valor |= GPIO_CTL_SET(PUSH_PULL_2MHZ, terminal);
    GPIOA(GPIO_CTL0) = valor;
}

void prender_led(int pin) {
    GPIOA(GPIO_BOP) = GPIO_BIT_MASK(terminal);
}
```

Usando punteros

- Se puede utilizar un *cast* para inicializar una variable de tipo puntero constante a la base del dispositivo
- Se puede utilizar la aritmética de punteros para acceder a todos los registros
  - Esto no es otra cosa que la representación en C del modo de direccionamiento indexado
- Cuando se suma una constante a un puntero tipado la constante se escala por el tamaño del tipo apuntado
  - Si el puntero es a enteros de 32 bits se debe redefinir los desplazamientos dividiendo en 4 los valores originales

La secuencia usando punteros

```
#define GPIOA_BASE      0x40010800 // Direccion base GPIOA
#define GPIO_CTL0       0          // Control de terminales 0 a 7
#define GPIO_BOP        4          // Puesta a 1 de las salidas
#define PUSH_PULL_2MHZ  0x02       // Salida push-pull de 2 MHz

#define GPIO_CTL_SET(value, pin)  (value << ((pin & 7) << 2))
#define GPIO_CTL_MASK(pin)        ~GPIO_CTL_SET(0x0F, pin)
#define GPIO_BIT_MASK(pin)        (1 << pin)
```

```
volatile unsigned int * const GPIOA = (volatile unsigned int *)GPIOA_BASE;

void configurar_led(int pin) {
    unsigned int valor = *(GPIOA + GPIO_CTL0);
    valor &= GPIO_CTL_MASK(terminal);
    valor |= GPIO_CTL_SET(GPIO_OUTPUT_2MHZ_PUSH_PULL, terminal);
    *(GPIOA + GPIO_CTL0) = valor;
}

void prender_led(int pin) {
    *(GPIOA + GPIO_BOP) = GPIO_BIT_MASK(terminal);
}
```

Usando vectores

- Un vector es un espacio de memoria contiguo que se puede acceder en base a un indice
- Cuando se accede a un elemento de un vector suma a la dirección de memoria del primer elemento el indice escalado por el tamaño de los elementos del vector
  - Esta operación es exactamente la misma que se realizo con los punteros en el caso anterior
- En el lenguaje C una variable puede utilizarse como puntero o como vector indistintamente

La secuencia usando vectores

```
#define GPIOA_BASE      0x40010800 // Direccion base GPIOA
#define GPIO_CTL0       0          // Control de terminales 0 a 7
#define GPIO_BOP        4          // Puesta a 1 de las salidas
#define PUSH_PULL_2MHZ  0x02       // Salida push-pull de 2 MHz

#define GPIO_CTL_SET(value, pin)  (value << ((pin & 7) << 2))
#define GPIO_CTL_MASK(pin)        ~GPIO_CTL_SET(0x0F, pin)
#define GPIO_BIT_MASK(pin)        (1 << pin)
```

```
volatile unsigned int * const GPIOA = (volatile unsigned int *)GPIOA_BASE;

void configurar_led(int pin) {
    unsigned int valor = GPIOA[GPIO_CTL0];
    valor &= GPIO_CTL_MASK(terminal);
    valor |= GPIO_CTL_SET(GPIO_OUTPUT_2MHZ_PUSH_PULL, terminal);
    GPIOA[GPIO_CTL0] = valor;
}

void prender_led(int pin) {
    GPIOA[GPIO_BOP] = GPIO_BIT_MASK(terminal);
}
```

Usando estructuras

- El acceso usando punteros tipados o vectores es útil solo si todos los registros del dispositivo son del mismo tamaño
  - Ya se discutió en temas anteriores que esto no es necesariamente así, muchos dispositivos tienen registros de diferentes tamaños
- Una opción es utilizar punteros tipo **`void`** y desplazamientos en bytes
- Las estructuras tambien son espacios de memoria contiguos pero con elementos de diferentes tipo
  - Una alternativa es definir una estructura donde los tipos coincidan con los tamaño de los registros y un puntero a esta estructura
  - Si hay varios dispositivos iguales en memoria contigua ahora se puede acceder como un vector de estas estructuras

La secuencia usando estructuras

```
typedef struct {
    volatile unsigned int CTL0;  // Control de terminales 0 a 7 a 7)
    ...
    volatile unsigned int BOP;   // Puesta a 1 de las salidas
    ...
} GPIO_TypeDef;

GPIO_TypeDef * const GPIOA = (GPIO_TypeDef *)GPIOA_BASE;
```

```
void configurar_led(int pin) {
    unsigned int valor = GPIOA->CTL0;
    valor &= GPIO_CTL_MASK(terminal);
    valor |= GPIO_CTL_SET(GPIO_OUTPUT_2MHZ_PUSH_PULL, terminal);
    GPIOA->CTL0 = valor;

}

void prender_led(int pin) {
    GPIOA->BOP = GPIO_BIT_MASK(terminal);
}
```

Controladores de dispositivos

- En los diferentes ejemplos el código de la aplicación principal no cambió
  - Esto se debe a que se definieron funciones para encapsular las operaciones con los dispositivos
- La implementación interna de estas funciones es independiente de la aplicación principal
  - Se puede cambiar la implementación interna sin afectar a la aplicación, lo que facilita moverla a otra plataforma: es portable
  - Tambien permite reutilizar las funciones que interactúan con los dispositivos en diferentes proyectos sobre el mismo hardware
- El conjunto de funciones definidas para interactuar con un dispositivo en particular es un controlador o driver

Usando la HAL del fabricante

- El fabricante del microcontrolador suele proveer una biblioteca para acceso a los dispositivos
- Esta biblioteca define variables para acceder a los registros de los dispositivos
  - El estilo varia según el fabricante, pero en general, será alguno de los que se mostraron en los ejemplos anteriores
  - En el caso de nuestro dispositivo se utilizan macros muy parecidos al primer ejemplo completo
- Esta biblioteca tambien suele incluir funciones para configurar y operar con los dispositivos sin necesidad de conocer los registros de los mismos
  - Suelen denominarla como HAL (Hardware Abstraction Layer)

Conclusiones

- El acceso a los dispositivos de entrada/salida es similar a la operación con variables en memoria
- Se puede realizar tanto en ensamblador como en cualquier lenguaje que soporte punteros
  - En el caso de C hay diferentes opciones
- Se pueden agrupar las operaciones sobre los dispositivos en funciones especificas
  - El conjunto de estas funciones es un controlador
- El uso de una capa de abstracción de hardware facilita:
  - El uso de los dispositivos sin conocer el detalle del funcionamiento
  - La reutilización en diferentes proyectos sobre la misma plataforma
  - La portabilidad de la aplicación a diferentes plataformas