UTN.BA

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

UNIVERSIDAD TECNOLÓGICA NACIONAL - FACULTAD REGIONAL BUENOS AIRES

Departamento de Ingeniería Electrónica

Augusto Santini

Librería LPC845 - Documentación

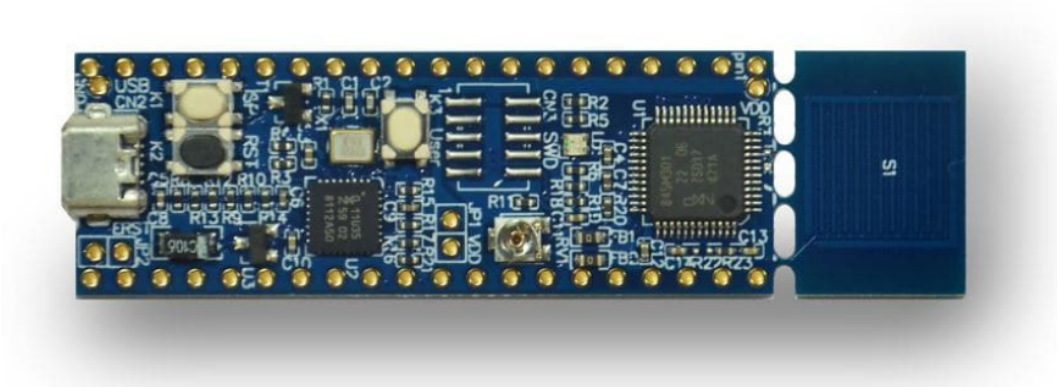

Tabla de contenidos

Capítulo 1

Introducción

Este documento está preparado como guía de consulta para el uso de la **Librería LPC845**. Dicha librería está pensada para un uso lo más flexible posible, únicamente solucionando problemas a nivel de periférico del microcontrolador **LPC845**, y no así niveles de abstracción mayores.

1.1. Organización de la librería

- *Hardware Register Interface*: Definición de registros e implementación de funciones a nivel registro para modificación de los mismos.
- *Hardware Proxy Layer*: Definición de funciones a nivel periférico para inicialización y manipulación de los mismos.

1.2. Utilización para su correcto funcionamiento

El usuario de la librería debería utilizar únicamente las funciones disponibles en la capa HPL (Hardware Proxy Layer). Los pasos típicos para poder utilizar la mayoría de los periférico suelen ser:

- *Inicialización*: Habilitación de clock y eventual encendido del periférico.
- *Clock*: Una vez habilitado el clock del periférico, debería configurarse los divisores que contenga el mismo para que opere a la frecuencia deseada.
- *Configuración*: Configuración propia del periférico para su correcta operación.
- *Interrupciones*: Habilitación/Inhabilitación de interrupciones tanto en el periférico como en el *NVIC*.

La librería se ocupa de todas estas inicializaciones al llamar a las distintas funciones de tipo *init* para los distintos periféricos disponibles. Nótese que en la librería fueron implementadas ciertas funcionalidades de cada periférico, las cuales suelen ser las más utilizadas.

Una vez llamada a la función de tipo *init* del periférico a utilizar, se pueden usar las distintas funciones para el acceso del mismo. Las implementaciónes de los distintos periféricos se explican en las subsiguientes secciones, con ejemplos adjuntos.

1.3. Interrupciones

La mayoría de los periféricos tienen la posibilidad de registrar llamados de función que se ejecutan desde interrupciones. Las funciones registradas en estos llamados **deben** ser lo más cortos

posibles, ya que una interrupción quita del flujo natural al programa principal. Una buena práctica teniendo esto en cuenta, es utilizar los llamados de función únicamente para habilitar ciertos flags, y luego en la aplicación principal consultar el estado de dichos flags y efectuar la acción que se necesite.

1.4. Ejemplos

Todos los ejemplos ilustrados en este documento, son implementables con el stick de desarrollo **LPC845-BRK**, sin necesidad de ningún componente externo adicional.

Capítulo 2

Analog to Digital Converter (ADC) - Conversor Analógico a Digital

El *ADC*, como su nombre lo indica, convierte una o más entradas analógicas, a un valor equivalente digital. En el caso del LPC845, tiene un único módulo *ADC* con una resolución de 12 bits, el cual tiene 12 canales, lo cual implica que se pueden realizar conversiones de 12 fuentes analógicas distintas, pero no así realizar conversiones *al mismo tiempo*. En caso de querer tomar señales de múltiples fuentes analógicas, se deberán hacer sucesivas conversiones en los distintos canales deseados.

Una resolución de 12 bits implica que la conversión aumentará cada unidad siguiendo la siguiente ecuación:

$$ADC_{res} = \frac{V_{ref_p}}{2^N}$$

Que en nuestro caso será:

$$ADC_{res} = \frac{3{,}3V}{2^{12}}$$
$$ADC_{res} \cong 805{,}66\mu V$$

Esto implica que podemos preveer el valor resultante de la conversión analógica/digital mediante la siguiente ecuación:

$$ADC_{conv} = \frac{V_{ADC_{in}}}{ADC_{res}}$$

Ejemplo: Teniendo una $V_{ADC_{in}} = 1V$ calculamos el resultado de la conversión:

$$ADC_{conv} = \frac{1V}{805{,}66\mu V}$$
$$ADC_{conv} = 1241{,}21$$

Pero el *ADC* convierte a valores enteros, por lo que redondeará, obteniendo:

$$ADC_{conv} = 1241$$

2.1. Concepto de *Secuencia de conversión*

Para el *ADC* de este microcontrolador, un inicio de conversión en realidad implica el inicio de una *secuencia de conversión*. Dicha secuencia puede implicar uno o más canales a convertir, y puede generar eventos tanto cuando se termina la secuencia completa, o cuando se termina cada canal de la secuencia. Asimismo los inicios de conversión pueden disparar una secuencia completa, o uno de los canales de dicha secuencia.

Se tienen dos secuencias configurables (Secuencia A y Secuencia B), las cuales se pueden configurar de forma tal que una secuencia interrumpa a la otra.

2.2. Inicio de conversiones

El *ADC* de este microcontrolador permite el inicio de secuencia de conversión de dos formas:

- Iniciadas por software: Se inician las secuencias de conversiones mediante código.
- Iniciadas por hardware: Se inician las secuencias de conversiones dependiendo de distintos periféricos del microcontrolador.

2.3. Calibración de hardware

Este periférico contiene un bloque de autocalibración, el cual debe ser utilizado luego de cada reinicio del microcontrolador o cada vez que se sale de modo de bajo consumo, para obtener la resolución y presición especificada por el fabricante.

2.4. Velocidad de conversión

Cada conversión realizada toma un tiempo que dependerá del clock configurado en el periférico. Podemos obtener este tiempo de conversión mediante la ecuación:

$$t_{conv_{ADC}} = \frac{1}{25 * f_{ADC}}$$

El multiplicador 25 en el denominador, es debido a la naturaleza del periférico de *aproximaciones sucesivas*. Esto implica que desde que se genera un inicio de conversión hasta que la misma finaliza, deben transcurrir 25 ciclos de clock del *ADC*.

Ejemplo: Configurando el *ADC* con una $f_{ADC} = 25MHz$ obtenemos el tiempo tomado por cada conversión:

$$t_{conv_{ADC}} = \frac{1}{25 * 1MHz}$$
$$t_{conv_{ADC}} = 1\mu s$$

Esto implica que entre un inicio de conversión y la finalización de la misma, pasará $1\mu s$. Nótese que este tiempo corresponde a una conversión para un único canal, en caso de estar convirtiendo varios canales, se deberá multiplicar $t_{conv_{ADC}}$ por la cantidad de canales activos en la secuencia de conversión, para obtener el tiempo total desde un inicio de secuencia de conversión y la finalización de todos los canales.

2.5. Implementación en la librería

- Conversiones de uno o múltiples canales.
- Conversiones únicas o en modo "ráfaga".
- Única secuencia de conversión.
- Inicio de secuencia de conversión únicamente por software.
- Un inicio de conversión dispara toda la secuencia de conversión.
- Se registra una función a ejecutar una vez terminada la secuencia de conversión.

2.5.1. Estructuras y enumeraciones asociadas

Configuración

```
typedef struct
{
        uint16_t channels;
        uint8_t burst;
        void (*conversion_ended_callback)(void);
}ADC_conversions_config_t;
```

Estructura para la configuración del ADC.

- channels: Canales habilitados para la conversión. Cada n bit representa al n canal. Ejemplo: Para habilitar los canales 0, 4 y 11 este parámetro debería valer 0b0000100000010001 = 0x0811.
- burst: Si este parámetro es cero, las conversiones son únicas. Cualquier otro valor habilita el modo "ráfaga".
- conversion_ended_callback: Puntero a función a ejecutar una vez terminada la secuencia de conversión.

2.5.2. Funciones asociadas

Inicialización

```
int32_t ADC_init(uint32_t adc_freq);
```

Función que se ocupa del encendido del periférico, habilitación del clock del mismo, calibración de hardware y fijar el clock asociado a la frecuencia deseada por el usuario.

Parámetros:

- adc_freq: Frecuencia de conversión deseada

Posibles valores de retorno:

- ADC_INIT_SUCCESS: Inicialización exitosa. El periférico realizó la calibración de hardware y se pudo fijar el clock deseado.
- ADC_INIT_CLK_OVERFLOW: La frecuencia de clock que se intentó fijar es demasiado alta (Mayor a 25MHz). No se realiza ningún tipo de configuración ni encendido del periférico.
- ADC_INIT_CLK_UNDERFLOW: La frecuencia de clock deseada es demasiado baja para poder fijarlo. No se realiza ningún tipo de configuración ni encendido del periférico.

Configuración

```
int32_t ADC_config_conversions(ADC_conversions_config_t *conversions_config);
```

Función que configura el ADC en base a la configuración pasada. Ver estructura 2.5.1. Si el mismo es configurado en modo "ráfaga", entonces las conversiones inician inmediatamente, caso contrario, deberán iniciarse por software. Si no se quiere configurar el periférico para que genere interrupciones, y se lo quiere consultar por pooling, el callback pasado en la estructura de configuracion deberá ser NULL.

Parámetros:

- conversions_config: Puntero a configuración de ADC. Para mas información, referirse a 2.5.1.

Posibles valores de retorno:

- ADC_CONFIG_CONVERSIONS_SUCCESS: La configuración del ADC fue exitosa.
- ADC_CONFIG_CONVERSIONS_NOT_POWERED: El periférico no fue alimentado. No se configura nada.
- ADC_CONFIG_CONVERSIONS_NOT_CLOCKED: El clock del periférico no fue habilitado. No se configura nada.
- ADC_CONFIG_CONVERSIONS_INVALID_CHANNELS: Se pasaron uno o más canales inválidos para la configuración del ADC. No se configura nada.

Inicio de secuencia de conversión

```
int32_t ADC_start_conversions(void);
```

El llamado a esta función implica iniciar una secuencia de conversión en base a la configuración lograda por las funciones explicadas en 2.5.2 y 2.5.2.

Parámetros:

Ninguno.

Posibles valores de retorno:

- ADC_START_CONVERSIONS_SUCCESS: El inicio de la secuencia de conversión fue exitoso. Cuando la secuencia de conversión finalice, se ejecutará el callback registrado mediante la función explicada en 2.5.2.
- ADC_START_CONVERSIONS_BURST_MODE: El ADC está configurado en modo "ráfaga" por lo que no tiene sentido iniciar una secuencia de conversión mediante software.
- ADC_START_CONVERSIONS_NOT_POWERED: El periférico no está alimentado. No hace nada.
- ADC_START_CONVERSIONS_NOT_CLOCKED: El periférico no tiene habilitado el clock. No hace nada.

Obtención de conversiones

```
int32_t ADC_get_conversion(uint8_t channel, uint32_t *conversion);
```

El llamado a esta función intentará recuperar el resultado de la conversión realizada en el canal *channel* (Valor de 0 a 11), guardando el resultado en *conversion*.

Parámetros:

- channel: Canal del cual recuperar el resultado. Valor entre 0 y 11. conversion: Puntero a donde guardar el resultado de la conversión si está disonible.

Posibles valores de retorno:

- ADC_GET_CONVERSION_SUCCESS: El resultado guardado en *conversion* es válido.
- ADC_GET_CONVERSION_NOT_POWERED: El periférico no está alimentado. No hace nada.
- ADC_GET_CONVERSION_NOT_CLOCKED: El clock del periférico no fue habilitado. No hace nada.
- ADC_GET_CONVERSION_INVALID_CHANNEL: El canal pasado es inválido (Mayor a 11). No hace nada.
- ADC_GET_CONVERSION_NOT_FINISHED: La secuencia de conversión configurada todavía no terminó. No hace nada.

Registrar callback de finalización de secuencia de conversión

```
void ADC_register_callback(void (*new_callback)(void));
```

Llamar a esta función con el parametro *new_callback* igual a NULL, implica inhabilitar las interrupciones para cuando se termina una secuencia de conversión. Si el parámetro no es NULL, se habilitan las interrupciones para cuando se termina una secuencia de conversión, y cada vez que finaliza una secuencia de conversión, se llama a dicho callback.

Capítulo 3

Digital to Analog Converter (DAC) - Conversor Digital Analógico

El *ADC*, como su nombre lo indica, convierte un valor digital, a un valor analógico. En el caso del LPC845, tiene dos módulos *DAC* con una resolución de 10 bits cada uno. Al igual que con el *ADC*, podemos calcular la resolución del *DAC* como:

$$DAC_{res} = \frac{V_{ref_p}}{2^N}$$

Que para nuestro caso será:

$$DAC_{res} = \frac{3{,}3V}{2^10}$$
$$DAC_{res} \cong 3{,}22mV$$

Esto implica que podemos preveer el valor de salida analógico sabiendo el valor colocado en el *DAC* con la siguiente ecuación:

$$DAC_{conv} = DAC_{res} * DAC_{val}$$

Ejemplo: Colocamos en el *DAC* el valor 372 para convertirlo a su equivalente analógico. En su correspondiente salida entonces tendremos:

$$DAC_{conv} = 3{,}22mV * 372$$
$$DAC_{conv} \cong 1{,}198mV$$

3.1. Velocidad de conversión

Este periférico permite dos velocidades de conversión para manejar el consumo de energía:

- 1Mhz mayor consumo de energía
- 400KHz menor consumo de energía

3.2. Implementación en la librería

- Velocidad de conversión configurable.
- Actualización de valor de salida por software.

3.2.1. Estructuras y enumeraciones asociadas

Ninguna.

3.2.2. Funciones asociadas

Inicialización

```
int32_t DAC_init(uint8_t dac_selection, uint8_t settling_time, uint32_t initial_value);
```

Función que se ocupa del encendido del periférico, habilitación del clock, y de fijar la velocidad de conversión del mismo.

Parámetros:

- dac_selection: Selección de instancia de DAC a inicializar.
- settling_time: Si este parámetro es cero, la instancia de DAC será inicializada con velocidad de conversión baja. Cualquier otro número inicializa la instancia de DAC en velocidad de conversión alta.
- initial_value: Valor inicial a poner en el DAC una vez inicializado.

Posibles valores de retorno:

- DAC_INIT_SUCCESS: La inicialización fue exitosa.
- DAC_INIT_INVALID_DAC: La instancia de DAC no existe. (dac_selection ¿1)

Actualización de valor

```
int32_t DAC_update_value(uint8_t dac_selection, uint32_t new_value);
```

Función que se encarga de actualizar el valor actual en la salida del DAC.

Parámetros:

- dac_selection: Selección de instancia de DAC a actualizar.
- new_value: Valor a actualizar en el DAC.

Posibles valores de retorno:

- DAC_UPDATE_SUCCESS: Actualización del valor exitosa.
- DAC_UPDATE_INVALID_DAC: La instancia de DAC no existe. (dac_selection ¿1)

Capítulo 4

General Purpouse Input/Output (GPIO) - Entradas/Salidas de Propósito General

El periférico *GPIO* es el encargado de controlar tanto entradas como salidas *digitales*. Esto implica que las salidas solamente podrán tomar como valores *cero* o *uno* y que las entradas únicamente interpretarán valores *cero* o *uno*.

Los pines en el LPC845 están descriptos mediante un *puerto* y un número de *pin*. Tiene dos *puertos* (0 y 1) con 32 *pines* cada uno (0 a 31). Cada uno de los pines del microcontrolador están conectados a uno de los pines de los conectores del stick de desarrollo.

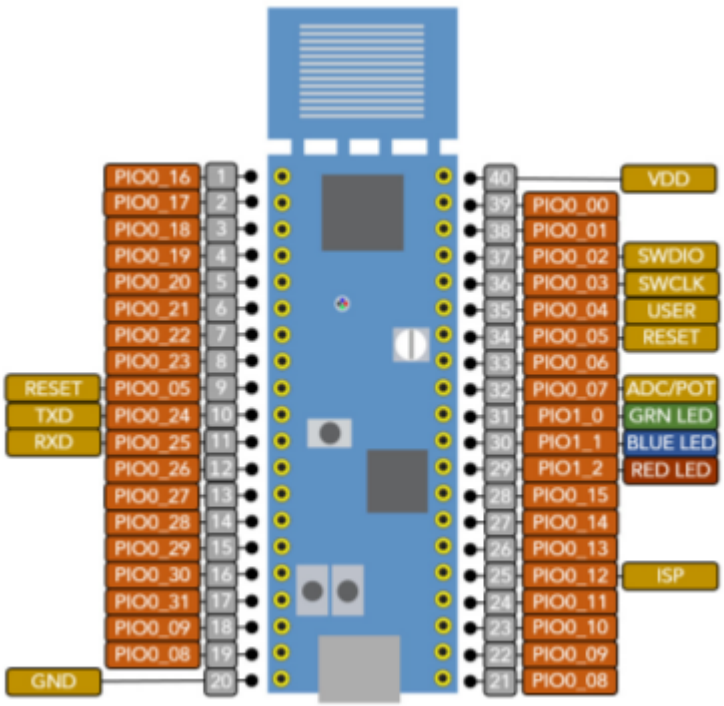

Figura 4.1: Conexiones de los pines del microcontrolador a cada uno de los conectores del stick de desarrollo

NOTA: El periférico *IOCON* está relacionado con este periférico. Leer la Sección 5 para más información.

4.1. Implementación en la librería

- Configuración de cada pin como entrada o salida.
- Manipulación de las salidas.
- Lectura de las entradas.

4.1.1. Estructuras y enumeraciones asociadas

Ninguna

4.1.2. Funciones asociadas

Inicialización

```
void GPIO_init(uint32_t port);
```

Función que se ocupa de inicializar un puerto del microcontrolador. Habilita el clock del puerto indicado.

Parámetros:

- port: Puerto a inicializar. Si se pasa cualquier número mayor a 1, la función no hace nada.

Posibles valores de retorno:

Ninguno.

Configuración

```
int32_t GPIO_set_dir(uint32_t port, uint32_t pin, uint8_t dir, uint8_t initial_state);
```

Función que se encarga de configurar la *dirección* (entrada/salida) del pin indicado.

Parámetros:

- port: Puerto del pin a configurar.
- pin: Número de pin del pin a configurar.
- dir: Dirección a configurar. GPIO_DIR_INPUT o GPIO_DIR_OUTPUT.
- initial_state: Estado inicial del pin. Sólo tiene sentido si el pin se configura como salida.

Posibles valores de retorno:

- GPIO_SET_DIR_SUCCESS: Configuración del pin exitosa.
- GPIO_SET_DIR_NOT_CLOCKED: El puerto asociado al pin no está inicializado.
- GPIO_SET_DIR_INVALID_PORT: El puerto pasado es inválido. ($port > 1$)
- GPIO_SET_DIR_INVALID_PIN: El número de pin pasado es inválido. ($pin > 31$)
- GPIO_SET_DIR_INVALID_PORTPIN: La combinación de puerto y pin no es válida para este microcontrolador. ($port = 1 \land pin > 21$)

Fijar estado alto de una salida

```
int32_t GPIO_set_pin(uint32_t port, uint32_t pin);
```

Función para actualizar el valor de una salida, poniendola en estado alto. Nota: Si el pin está configurado como entrada, no se verá reflejado el estado en el mismo.

Parámetros:

- port: Puerto del pin a actualizar.
- pin: Número de pin del pin a actualizar.

Posibles valores de retorno:

- GPIO_SET_PIN_SUCCESS: El pin fue accionado correctamente.
- GPIO_SET_PIN_NOT_CLOCKED: El puerto asociado no está inicializado.
- GPIO_SET_PIN_INVALID_PORT: El puerto pasado es inválido. ($port > 1$)
- GPIO_SET_PIN_INVALID_PIN: El número de pin pasado es inválido. ($pin > 31$)
- GPIO_SET_PIN_INVALID_PORTPIN: La combinación de puerto y pin no es válida para este microcontrolador. ($port = 1 \wedge pin > 21$)

Fijar estado bajo de una salida

```
int32_t GPIO_clear_pin(uint32_t port, uint32_t pin);
```

Función para actualizar el valor de una salida, poniendola en estado bajo. Nota: Si el pin está configurado como entrada, no se verá reflejado el estado en el mismo.

Parámetros:

- port: Puerto del pin a actualizar.
- pin: Número de pin del pin a actualizar.

Posibles valores de retorno:

- GPIO_CLEAR_PIN_SUCCESS: El pin fue accionado correctamente.
- GPIO_CLEAR_PIN_NOT_CLOCKED: El puerto asociado no está inicializado.
- GPIO_CLEAR_PIN_INVALID_PORT: El puerto pasado es inválido. ($port > 1$)
- GPIO_CLEAR_PIN_INVALID_PIN: El número de pin pasado es inválido. ($pin > 31$)
- GPIO_CLEAR_PIN_INVALID_PORTPIN: La combinación de puerto y pin no es válida para este microcontrolador. ($port = 1 \wedge pin > 21$)

Invertir el estado de una salida

```
int32_t GPIO_toggle_pin(uint32_t port, uint32_t pin);
```

Función para actualizar el valor de una salida, invirtiendo su estado actual. Nota: Si el pin está configurado como entrada, no se verá reflejado el estado en el mismo.

Parámetros:

- port: Puerto del pin a actualizar.
- pin: Número de pin del pin a actualizar.

Posibles valores de retorno:

- GPIO_TOGGLE_PIN_SUCCESS: El pin fue accionado correctamente.
- GPIO_TOGGLE_PIN_NOT_CLOCKED: El puerto asociado no está inicializado.
- GPIO_TOGGLE_PIN_INVALID_PORT: El puerto pasado es inválido. ($port > 1$)
- GPIO_TOGGLE_PIN_INVALID_PIN: El número de pin pasado es inválido. ($pin > 31$)
- GPIO_TOGGLE_PIN_INVALID_PORTPIN: La combinación de puerto y pin no es válida para este microcontrolador. ($port = 1 \land pin > 21$)

Leer el estado de una entrada

```
int32_t GPIO_read_pin(uint32_t port, uint32_t pin);
```

Función para actualizar el valor de una salida, invirtiendo su estado actual.

Parámetros:

- port: Puerto del pin a leer.
- pin: Número de pin del pin a leer.

Posibles valores de retorno:

- 0: El pin no se encuentra activo.
- 1: El pin se encuentra activo.
- GPIO_READ_PIN_NOT_CLOCKED: El puerto asociado no está inicializado.
- GPIO_READ_PIN_INVALID_PORT: El puerto pasado es inválido. ($port > 1$)
- GPIO_READ_PIN_INVALID_PIN: El número de pin pasado es inválido. ($pin > 31$)
- GPIO_READ_PIN_INVALID_PORTPIN: La combinación de puerto y pin no es válida para este microcontrolador. ($port = 1 \land pin > 21$)

Capítulo 5

Input/Output Configuracion (IOCON) - Configuración de Entradas/Salidas

La configuración de los pines del microcontrolador son controlados por este módulo. Dichas configuraciones incluyen resistencias internas de *pull-up* y *pull-down*, configurar salidas del tipo *open drain*, *histéresis* del pin, *filtro de glitches* y *modo IIC*.

5.1. Inversión de pin

Este periférico tiene la posibilidad de invertir la lógica del pin, sea entrada o salida, mediante hardware. Asumiendo que no está activada la inversión del pin y que está configurado como entrada, si se toma una lectura del mismo, se leerá como **cero** cuando externamente esté a una tensión de aproximadamente V_{SS} y se leerá como **uno** cuando externamente esté a una tensión de aproximadamente V_{DD}, inversamente, si el pin está configurado como salida, al actualizarla a **cero** colocará en el pin una tensión de aproximadamente V_{SS} y al actualizarla a **uno** colocará en el pin una tensión de aproximadamente V_{DD}. Si se activa la inversión del pin sucederá lo inverso, es decir, si está configurado como entrada y externamente hay una tensión de aproximadamente V_{SS} la lectura se tomará como **uno** y si hay una tensión de aproximadamente V_{DD} la lectura se tomará como **cero**, y cuando se configure el pin como salida, al actualizarla en **cero** colocará una tensión de aproximadamente V_{DD} y al actualizarla en **uno** colocará una tensión de aproximadamente V_{SS}.

5.2. Resistencias internas

Cada uno de los pines del microcontrolador puede ser configurado para tener una resistencia de *pull-up* o de *pull-down*. Esto implica que aunque la entrada no tenga ningún valor fijo externamente, estas resistencias se encargarán de fijar un valor o bien naturalmente alto (*pull-up*) o bajo (*pull-down*). Existe un tercer tipo llamado *repetidor* el cual configura un *pull-up* o un *pull-down* dependiendo del último valor que hubo en la entrada, esto quiere decir que si el pin tiene un valor externamente fijado en estado alto, y queda sin un valor fijo, el microcontrolador automáticamente dejará configurado un *pull-up*, de esta forma reteniendo el último valor fijo, y viceversa cuando el pin es fijado externamente en estado bajo.

5.3. Salidas de tipo open-drain

Una salida tipo *open drain* en comparación con una salida tradicional, permite colocar un nivel de tensión distinto al de la alimentación del microcontrolador cuando se acciona el pin. **NOTA**: La hoja de datos aclara que la tensión a la cual se puede manejar el pin mediante un resistor de *pull-up* externo, no puede superar la tensión V_{DD}.

5.4. Histéresis

Cada pin puede configurar una *histéresis* cuando se comporta como entrada. Que el pin tenga una *histéresis* implica que el valor cambiará su lectura una vez superado un cierto umbral V_{hys}. Esto evita lecturas erróneas cuando la entrada tiene una variación que no se puede evitar.

5.5. Filtro de glitches

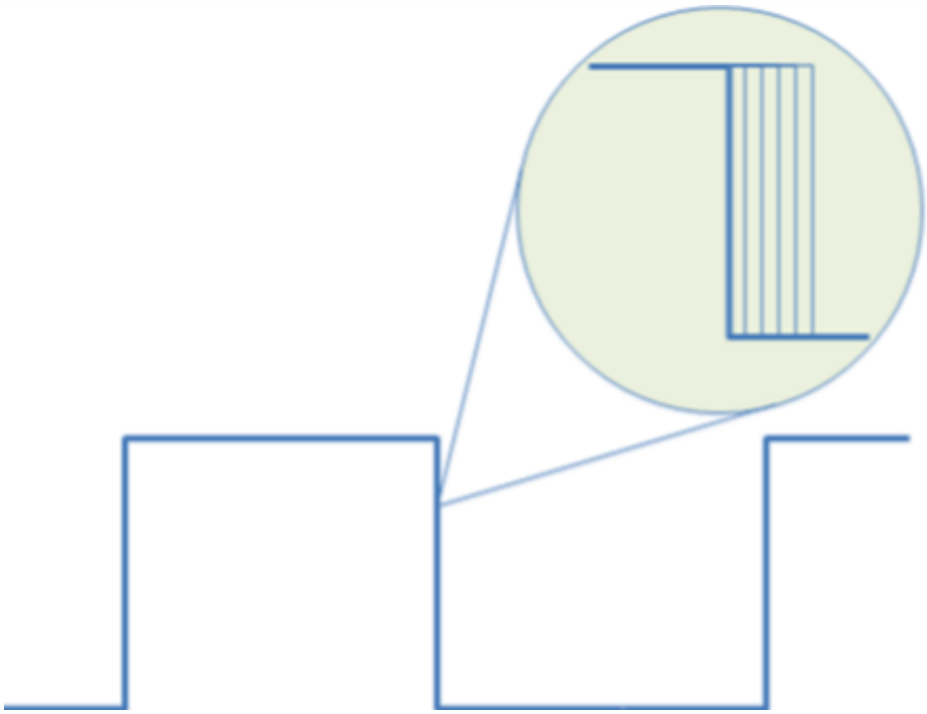

Figura 5.1: Entrada con glitches o rebotes

Si una entrada tiene *glitches* o *rebotes*, como se muestra en la Figura 5.1, se pueden tomar lecturas erróneas, a causa de la falta de estabilidad de la misma. Una forma de evitar este problema es configurando correctamente el clock asociado al periférico *IOCON*, y el filtro de *glitches* asociado al pin, el cual tomará una lectura como válida, luego de una cantidad configurable de ciclos de clock.

5.6. Modo IIC

Cuando se utiliza el periférico *IIC* los pines tienen una configuración por hardware particular, la cual es lograda mediante esta configuración.

5.7. Implementación en la librería

- Configuración de resistencias internas.
- Configuración de modo open drain.
- Habilitación/Inhabilitación de histéresis.
- Configuración del filtro de glitch.

5.7.1. Estructuras y enumeraciones asociadas

Configuración de resistencia interna

```
typedef enum
{
        PULL_NONE = 0,
        PULL_DOWN,
        PULL_UP,
        PULL_REPEATER
}IOCON_pull_mode_en;
```

Enumeraciones para configuración de resistencia interna de los pines. **NOTA**: Los pines que admiten función I2C, no tienen resistencia interna, por lo que este campo no tiene efecto alguno. En caso de necesitar un pull-up o pull-down, se deberá colocar externamente.

- PULL_NONE: El pin no tiene ni pull-up ni pull-down interno.
- PULL_DOWN: El pin tiene un pull-down interno.
- PULL_UP: El pin tiene un pull-up interno.
- PULL_REPEATER: El pin tiene una resistencia interna tipo repetidora.

Configuración de filtrado

```
typedef enum
{
        SAMPLE_MODE_BYPASS = 0,
        SAMPLE_MODE_1_CLOCK,
        SAMPLE_MODE_2_CLOCK,
        SAMPLE_MODE_3_CLOCK
}IOCON_sample_mode_en;
```

Enumeraciones para la configuración del filtro de glitches del pin.

- SAMPLE_MODE_BYPASS: No se utiliza el filtro de glitches. La entrada es reflejada de manera inmediata.
- SAMPLE_MODE_1_CLOCK: La entrada se refleja por cada ciclo de clock del periférico.
- SAMPLE_MODE_2_CLOCK: La entrada se refleja por cada 2 ciclos de clock del periférico.
- SAMPLE_MODE_3_CLOCK: La entrada se refleja por cada 3 ciclos de clock del periférico.

Selección del clock para el pin

```
typedef enum
{
        IOCON_CLK_DIV_0 = 0,
        IOCON_CLK_DIV_1,
        IOCON_CLK_DIV_2,
        IOCON_CLK_DIV_3,
        IOCON_CLK_DIV_4,
        IOCON_CLK_DIV_5,
        IOCON_CLK_DIV_6
}IOCON_clk_sel_en;
```

Enumeraciones para la selección del clock para el pin.

- IOCON_CLK_DIV_0: Selección de CLKDIV0 para el pin.
- IOCON_CLK_DIV_1: Selección de CLKDIV1 para el pin.
- IOCON_CLK_DIV_2: Selección de CLKDIV2 para el pin.
- IOCON_CLK_DIV_3: Selección de CLKDIV3 para el pin.
- IOCON_CLK_DIV_4: Selección de CLKDIV4 para el pin.
- IOCON_CLK_DIV_5: Selección de CLKDIV5 para el pin.
- IOCON_CLK_DIV_6: Selección de CLKDIV6 para el pin.

Selección de modo I2C

```
typedef enum
{
        IIC_MODE_STANDARD = 0,
        IIC_MODE_GPIO,
        IIC_MODE_FAST_MODE
}IOCON_iic_mode_en;
```

Enumeraciones para la selección de modo de I2C en el pin. **NOTA**: No todos los pines admiten esta configuracion, revisar la documentación del microcontrolador para más información.

- IIC_MODE_STANDARD: Modo I2C standard.
- IIC_MODE_GPIO: El pin funcionará en modo GPIO.
- IIC_MODE_FAST_MODE: modo I2C de alta velocidad.

Configuración de pin

```
typedef struct
{
        IOCON_pull_mode_en mode;
        uint8_t hysteresis;
        uint8_t invert_input;
        uint8_t open_drain;
        IOCON_sample_mode_en sample_mode;
        IOCON_clk_sel_en clk_sel;
        uint8_t dac_mode;
        IOCON_iic_mode_en iic_mode;
}IOCON_config_t;
```

Estructura para configuración de un pin.

- mode: Configuración de resistencia interna.
- hysteresis: En caso de ser cero, no se activará la histéresis en el pin, caso contrario se activará.
- invert_input: En caso de ser cero, no se activará la inversión en el pin, caso contrario se activará.
- open_drain: En caso de ser cero, el pin trabajará como salida *totem pole*, caso contrario trabajará en modo *open drain*.
- sample_mode: Modo de sampleo del pin.
- dac_mode: En caso de ser cero, el pin funciona en modo normal, caso contrario se habilita la funcionalidad de DAC en el pin. **NOTA**: No todos los pines tienen esta funcionalidad, para más información, consultar el manual del microcontrolador.
- iic_mode: Configuración del modo I2C.

5.7.2. Funciones asociadas

Inicialización

```
void IOCON_init(void);
```

Función encargada de inicializar el periférico. Simplemente habilita el clock del periférico.

Parámetros:

Ninguno.

Posibles valores de retorno:

Ninguno.

Configuración

```
int32_t IOCON_config_io(uint32_t port, uint32_t pin, IOCON_config_t *pin_config);
```

Función para configurar un pin.

Parámetros:

- port: Puerto del pin a configurar.
- pin: Número de pin del pin a configurar.
- pin_config: Puntero a la configuración del pin.

Posibles valores de retorno:

- IOCON_CONFIG_SUCCESS: Configuración del pin exitosa.
- IOCON_CONFIG_IOCON_NOT_CLOCKED: El periférico no tiene habilitado el clock. La función no hace nada.
- IOCON_CONFIG_INVALID_PORT: El puerto pasado es inválido. ($port > 1$)
- IOCON_CONFIG_INVALID_PIN: El número de pin pasado es inválido. ($pin > 31$)
- IOCON_CONFIG_INVALID_PORTPIN: La combinación de puerto y pin no es válida para este microcontrolador. ($port = 1 \land pin > 21$)

Capítulo 6

Pin Interrupt (PININT) - Interrupciones de pin

El periférico *PININT*, permite detectar cambios en los pines sin intervención de software por parte del microcontrolador. En el caso de este microcontrolador, se tienen ocho canales distintos para detectar interrupciones de pin independientes, y cualquier pin se puede conectar con cualquiera de los ocho módulos *PININT* disponibles.

El periférico se puede configurar para generar interrupciones tanto cuando se detecta un flanco ascendente/descendente configurable por software, o cuando se detecta un nivel bajo/alto también configurable por software. Permite interrupciones tanto para uno de los dos flancos/niveles como para cualquiera de los dos.

NOTA: El periférico *IOCON* está relacionado con este periférico. Leer la Sección 5 para más información.

6.1. Implementación en la librería

- Configuración de cualquiera de los ocho canales del módulo.
- Configuración tanto por flanco como por nivel.
- Configuración para interrupciones en flanco descendente/nivel bajo como para flanco ascendente/nivel alto.

6.1.1. Estructuras y enumeraciones asociadas

Configuración

```
typedef struct
{
        uint32_t port;
        uint32_t pin;
        uint8_t interrupt;
        uint8_t mode;
        uint8_t int_on_rising_edge;
        uint8_t int_on_falling_edge;
        void (*int_callback)(void);
}PININT_config_t;
```

Estructura de configuración para la interrupción de pin.

- port: Puerto del pin al cual vincular la interrupción de pin.
- pin: Número de pin del pin al cual vincular la interrupción de pin.
- interrupt: Número de interrupción a configurar.
- mode: Modo de interrupción de pin, por flanco o nivel. Valores posibles: PININT_MODE_EDGE_SENSITIVE o PININT_MODE_LEVEL_SENSITIVE.
- int_on_rising_edge: En caso de ser cero, no estarán habilitadas las interrupciones en flanco ascendente/nivel alto, cualquier otro valor habilitará dichas interrupciones.
- int_on_falling_edge: En caso de ser cero, no estarán habilitadas las interrupciones en flanco descendente/nivel bajo, cualquier otro valor habilitará dichas interrupciones.
- int_callback: Puntero a función a ejecutar cuando sucedan los sucesos configurados. En caso de que se quieran deshabilitar las interrupciones, se puede pasar este puntero como NULL.

6.1.2. Funciones asociadas

Inicialización

```
void PININT_init(void);
```

Función encargada de inicializar el periférico. Únicamente habilita el clock del mismo.

Parámetros:

Ninguno.

Posibles valores de retorno:

Ninguno.

Configuración

```
int32_t PININT_configure_pin_interrupt(PININT_config_t *config);
```

Función encargada de configurar la interrupción.

Parámetros:

- config: Puntero a la estructura de configuración de la interupción de pin. Para más información, referirse a 6.1.1.

Posibles valores de retorno:

- PININT_CONFIGURE_SUCCESS: Configuración exitosa.
- PININT_CONFIGURE_INVALID_PORT: El puerto pasado es inválido. ($port > 1$)
- PININT_CONFIGURE_INVALID_PIN: El número de pin pasado es inválido. ($pin > 31$)
- PININT_CONFIGURE_INVALID_PORTPIN: La combinación de puerto y pin no es válida para este microcontrolador. ($port = 1 \land pin > 21$)
- PININT_CONFIGURE_NOT_CLOCKED: El periférico no tiene el clock habilitado.
- PININT_CONFIGURE_INVALID_INTERRUPT: El número de interrupción pasada es inválido. ($interrupt > 7$)

Registrar callback de interrupción

```
void PININT_register_callback(void (*new_callback)(void), uint32_t interrupt);
```

Función para registrar un callback a ser llamado cuando ocurran los sucesos configurados.

Parámetros:

- new_callback: Puntero a función a ser ejecutado cuando sucedan los sucesos configurados.
- interrupt: Número de interrupcion a la cual registrar el nuevo callback. En caso de querer deshabilitar las interrupciones, se puede pasar NULL.

Posibles valores de retorno:

- PININT_REGISTER_CALLBACK_SUCCESS: Se configuró correctamente el callback nuevo.
- PININT_REGISTER_CALLBACK_INVALID_INTERRUPT: El número de interrupción pasada es inválido. ($interrupt > 7$)
- PININT_REGISTER_CALLBACK_NOT_CLOCKED: El periférico no tiene el clock habilitado.

Capítulo 7

System Tick Timer (SYSTICK) - Timer de Ticks del Sistema

El periférico *SYSTICK* tiene como finalidad tener una base de tiempos fija para tareas que necesiten una cierta periodicidad en sus llamados. Cualquier tipo de servicio que necesite ser revisado o atendido frecuentemente, puede ser avisado mediante un flag con ayuda de este periférico.

La frecuencia de los ticks de sistema es configurable y en general se suelen utilizar valores alrededor del milisegundo. Es dependiente de la frecuencia de trabajo del microcontrolador.

7.1. Implementación en la librería

- Configuración de tiempo de tick

7.1.1. Estructuras y enumeraciones asociadas

Ninguna.

7.1.2. Funciones asociadas

Inicialización

```
void SYSTICK_init(uint32_t tick_us, void (*systick_hook)(void));
```

Función encargada de configurar el *SYSTICK* e inicializar el mismo. En caso de querer generar ticks demasiado lentos, se fijará al menor valor posible con el clock configurado.

Parámetros:

- tick_us: Tiempo deseado a configurar entre cada tick en microsegundos.
- systick_hook: Puntero a función a llamar en cada tick del sistema.

Posibles valores de retorno:

Ninguno.

Registrar callback

```
void SYSTICK_update_callback(void (*new_callback)(void));
```

Función para registrar el llamado a efectuar en cada tick del sistema.

Parámetros:

- new_callback: Puntero a función a ejecutar en cada tick del sistema.

Posibles valores de retorno:

Ninguno.

Capítulo 8

Universal Asynchronous Receiver Transmiter (Receptor/Transmisor Asincrónico Universal

El periférico *UART* permite comunicaciones de tipo *serie*, como su nombre lo indica, para la transferencia de datos entre dos (o más) dispositivos. En particular, este periférico no necesita una línea de clock para compartir entre los dos dispositivos, a diferencia de otro tipo de comunicaciones, por lo que ambos dispositivos deberán "ponerse de acuerdo" respecto a la velocidad de comunicación.

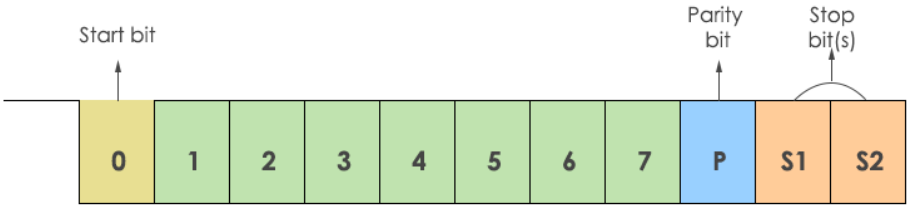

Figura 8.1: Trama de una comunicación UART

En la Figura 8.1 se muestra una trama de comunicación tipo *UART*. Nótese que hay ciertos bits que **deben** tener cierto valor. Dichos bits son:

- Bit de start (amarillo): Siempre es cero y es un único bit.
- Bits de datos (verde): Tomarán distintos valores dependiendo del mensaje que se quiera enviar. La cantidad de bits puede también ser variable, generalmente siendo 7, 8 o 9 bits.
- Bit de paridad (azul): Este bit puede encontrarse o no en la trama. En caso de encontrarse, sirve como método básico de chequeo de errores.
- Bit/s de stop (rojo): Su longitud es variable, siendo casos típicos 1 o 2 bits de stop. Su valor es siempre uno.

8.1. Concepto de *baudrate*

Como se explicó previamente, todas las partes implicadas en la comunicación, tienen que coincidir en la velocidad de transmisión de cada bit. A esta velocidad se la llama *baudrate* y se expresa en bits por segundo (*bps*). Valores típicos son: 9600bps, 19200bps, 115200bps. A mayor *baudrate*, toma menos tiempo enviar un dato de un punto a otro, lo cual puede ser crítico para ciertas aplicaciones.

8.2. Implementación en la librería

- Configuración de baudrate.
- Configuración de cantidad de bits de datos.
- Configuración de paridad.
- Configuración de bits de stop.
- Interrupción cuando llegó un dato nuevo.
- Interrupción cuando se terminó de enviar un dato.

8.2.1. Estructuras y enumeraciones asociadas

Configuración de oversampling

```
typedef enum
{
        UART_OVERSAMPLING_X5 = 4,
        UART_OVERSAMPLING_X6,
        UART_OVERSAMPLING_X7,
        UART_OVERSAMPLING_X8,
        UART_OVERSAMPLING_X9,
        UART_OVERSAMPLING_X10,
        UART_OVERSAMPLING_X11,
        UART_OVERSAMPLING_X12,
        UART_OVERSAMPLING_X13,
        UART_OVERSAMPLING_X14,
        UART_OVERSAMPLING_X15,
        UART_OVERSAMPLING_X16,
}UART_oversampling_en;
```

Enumeraciones para la configuración del oversampling en cada bit de la transmisión.

- UART_OVERSAMPLING_X5: Se samplea 5 veces cada bit.
- UART_OVERSAMPLING_X6: Se samplea 6 veces cada bit.
- UART_OVERSAMPLING_X7: Se samplea 7 veces cada bit.
- UART_OVERSAMPLING_X8: Se samplea 8 veces cada bit.
- UART_OVERSAMPLING_X9: Se samplea 9 veces cada bit.
- UART_OVERSAMPLING_X10: Se samplea 10 veces cada bit.
- UART_OVERSAMPLING_X11: Se samplea 11 veces cada bit.
- UART_OVERSAMPLING_X12: Se samplea 12 veces cada bit.
- UART_OVERSAMPLING_X13: Se samplea 13 veces cada bit.
- UART_OVERSAMPLING_X14: Se samplea 14 veces cada bit.
- UART_OVERSAMPLING_X15: Se samplea 15 veces cada bit.
- UART_OVERSAMPLING_X16: Se samplea 16 veces cada bit.

Configuración del largo de bits en los datos

```
typedef enum
{
        UART_DATALEN_7BIT = 0,
        UART_DATALEN_8BIT,
        UART_DATALEN_9BIT
}UART_datalen_en;
```

Enumeraciones para la configuración de la cantidad de bits en cada dato transmitido o recibido.

- UART_DATALEN_7BIT: Cada transmisión/recepción tendrá 7 bits de datos.
- UART_DATALEN_8BIT: Cada transmisión/recepción tendrá 8 bits de datos.
- UART_DATALEN_9BIT: Cada transmisión/recepción tendrá 9 bits de datos.

Configuración de paridad

```
typedef enum
{
        UART_PARITY_NO_PARITY = 0,
        UART_PARITY_EVEN = 2,
        UART_PARITY_ODD
}UART_parity_en;
```

Enumeraciones para la configuración de la paridad en la transmisión/recepción.

- UART_PARITY_NO_PARITY: Los mensajes no tendrán paridad.
- UART_PARITY_EVEN: Los mensajes tendrán paridad par.
- UART_PARITY_ODD: Los mensajes tendrán paridad impar.

Configuración de cantidad de bits de stop

```
typedef enum
{
        UART_STOPLEN_1BIT = 0,
        UART_STOPLEN_2BIT
}UART_stop_en;
```

Enumeraciones para la configuración de la cantidad de bits en la transmisión/recepción.

- UART_STOPLEN_1BIT: Los mensajes tendrán un bit de stop.
- UART_STOPLEN_2BIT: Los mensajes tendrán dos bit de stop.

Configuración

```
typedef struct
{
        uint8_t data_length;
```

```
        uint8_t parity;
        uint8_t stop_bits;
        uint8_t oversampling;
        uint32_t baudrate;
        uint32_t tx_port;
        uint32_t tx_pin;
        uint32_t rx_port;
        uint32_t rx_pin;
        void (*rx_ready_callback)(void);
        void (*tx_ready_callback)(void);
}UART_config_t;
```

Estructura de configuración de UART.

- data_length: Cantidad de bits de datos en la comunicación.
- parity: Configuración de paridad.
- stop_bits: Configuración de bits de stop.
- oversampling: Configuración de oversampling en cada bit.
- baudrate: Baudrate deseado para las comunicaciones.
- tx_port: Puerto asociado al pin de transmisión.
- tx_pin: Número de pin asociado al pin de transmisión.
- rx_port: Puerto asociado al pin de recepción.
- rx_pin: Número de pin asociado al pin de recepción.
- rx_ready_callback: Puntero a función a ejecutar cuando se recibe un dato.
- tx_ready_callback: Puntero a función a ejecutar cuando se termina de enviar un dato.

8.2.2. Funciones asociadas

Inicialización

```
int32_t UART_init(uint8_t uart_selection, UART_config_t *config);
```

Función para inicializar alguna instancia de UART. Se encarga de energizar el periférico, habilitar el clock asociado, y configurar la instancia en base a la configuración deseada.

Parámetros:

- uart_selection: Instancia de UART a inicializar.
- config: Puntero a la configuración deseada para la instancia de UART. Para más información, referirse a 8.2.1.

Posibles valores de retorno:

- UART_INIT_SUCCESS: La configuración de la instancia de UART fue exitosa.
- UART_INIT_INVALID_UART: La instancia de UART pasada es inválida. ($uart\_selection > 4$)
- UART_INIT_INVALID_PORT: El puerto pasado es inválido. ($port > 1$)

- UART_INIT_INVALID_PIN: El número de pin pasado es inválido. ($pin > 31$)
- UART_INIT_INVALID_PORTPIN: La combinación de puerto y pin no es válida para este microcontrolador. ($port = 1 \land pin > 21$)
- UART_INIT_CLOCK_UNDERFLOW: El baudrate deseado no es posible de configurar.

Transmitir un dato

```
int32_t UART_tx_byte(uint32_t uart_selection, uint32_t data);
```

Función para iniciar una transmisión de un dato.

Parámetros:

- uart_selection: Instancia de UART a través de la cual iniciar la transmisión.
- data: Dato a transmitir.

Posibles valores de retorno:

- UART_TX_BYTE_SUCCESS: El inicio de transmisión fue exitoso.
- UART_TX_INVALID_UART: La instancia de UART pasada es inválida. ($uart\_selection > 4$)
- UART_TX_NOT_AVAILABLE: La instancia de UART ya está con una transmisión pendiente.
- UART_TX_INVALID_BITS: Hay un error en la configuración de la UART, particularmente en la cantidad de bits de cada mensaje.

Recibir un dato

```
int32_t UART_rx_byte(uint32_t uart_selection, uint32_t *data);
```

Función para recibir un dato de una instancia de UART. En general, se llama a esta función desde el callback de interrupción de recepción lista.

Parámetros:

- uart_selection: Instancia de UART a través de la cual iniciar la transmisión.
- data: Puntero a donde guardar el dato recibido en caso de que esté disponible.

Posibles valores de retorno:

- UART_RX_BYTE_SUCCESS: La recepción del dato fue exitosa y el valor fue copiado al puntero pasado.
- UART_RX_INVALID_UART: La instancia de UART pasada es inválida. ($uart\_selection > 4$)
- UART_RX_NOT_AVAILABLE: La instancia de UART no tiene ningún dato pendiente para leer.
- UART_RX_INVALID_BITS: Hay un error en la configuración de la UART, particularmente en la cantidad de bits de cada mensaje.

Registrar callback de transmisión terminada

```
void UART_register_tx_callback(uint32_t uart_selection, void (*new_callback)(void));
```

Función para registrar un callback a ser llamado cuando se termine la transmisión de un mensaje.

Parámetros:

- uart_selection: Instancia de UART a través de la cual iniciar la transmisión.
- new_callback: Puntero a función a ejecutar cuando se termine la transmisión de un mensaje. En caso de querer deshabilitar este callback, se puede pasar NULL.

Posibles valores de retorno:

Ninguno.

Registrar callback de recepción terminada

```
void UART_register_rx_callback(uint32_t uart_selection, void (*new_callback)(void));
```

Función para registrar un callback a ser llamado cuando se termine la recepción de un mensaje.

Parámetros:

- uart_selection: Instancia de UART a través de la cual iniciar la transmisión.
- new_callback: Puntero a función a ejecutar cuando se termine la recepción de un mensaje. En caso de querer deshabilitar este callback, se puede pasar NULL.

Posibles valores de retorno:

Ninguno.

8.3. Ejemplos

En esta sección se presentan diversos ejemplos de utilización de la librería expuesta en este documento. Cualquiera de los ejemplos que se desee implementar en *MCUXpresso* deberá tener incluida correctamente la librería para su correcto funcionamiento.

8.3.1. Ejemplo 1 - Blink del LED verde del stick

Funcionamiento

El programa cambiará el estado del pin asociado al LED verde del stick de desarrollo cada un segundo.

Explicación

En primer lugar se inicializan los periféricos a utilizar. En este caso: *GPIO* y *SYSTICK*. Del periférico *GPIO* se configura únicamente el pin asociado al LED verde del stick, y el *SYSTICK* se configura para generar interrupciones cada $1000\mu seg = 1mseg$. En la función llamada en cada interrupción de *SYSTICK* se incrementa un contador, el cual al llegar a mil cuentas se reinicia y cambia el estado del LED verde.

Código

```
#include <HPL_GPIO.h>
#include <HPL_SYSTICK.h>

#define         RED_LED_PORT        1
#define         RED_LED_PIN         0

#define         RED_LED_OFF_STATE   1
#define         RED_LED_ON_STATE    0

static void tick_int(void);

/*
 * @brief   Application entry point.
 */
int main(void) {
        GPIO_init(1);

        GPIO_set_dir(RED_LED_PORT,
                     RED_LED_PIN,
                     GPIO_DIR_OUTPUT,
                     RED_LED_OFF_STATE);

        SYSTICK_init(1000, tick_int);

        while(1)
        {
                // Do nothing.
        }

    return 0 ;
}

static void tick_int(void)
{
        static uint32_t contador = 0;

        contador++;
        contador %= 1000;

        if(!contador)
        {
                GPIO_toggle_pin(RED_LED_PORT, RED_LED_PIN);
        }
}
```

8.3.2. Ejemplo 2 - Ciclo de encendido de cada LED RGB del stick

Funcionamiento

El programa cambiará el estado del pin asociado al LED verde del stick de desarrollo cada un segundo.

Explicación

En primer lugar se inicializan los periféricos a utilizar. En este caso: *GPIO* y *SYSTICK*. Del periférico *GPIO* se configura únicamente el pin asociado al LED verde del stick, y el *SYSTICK* se configura para generar interrupciones cada $1000\mu seg = 1mseg$. En la función llamada en cada interrupción de *SYSTICK* se incrementa un contador, el cual al llegar a mil cuentas se reinicia y cambia el estado del LED verde.

Código

```
1  #include <HPL_GPIO.h>
2  #include <HPL_SYSTICK.h>
3
4  #define        RED_LED_PORT                1
5  #define        RED_LED_PIN                       2
6
7  #define        GREEN_LED_PORT               1
8  #define        GREEN_LED_PIN                0
9
10 #define        BLUE_LED_PORT                1
11 #define        BLUE_LED_PIN                1
12
13 #define        LED_OFF_STATE                1
14 #define        LED_ON_STATE                0
15
16 #define        RED_LED_ON                        GPIO_clear_pin(RED_LED_PORT, RED_LED_PIN)
17 #define        RED_LED_OFF                        GPIO_set_pin(RED_LED_PORT, RED_LED_PIN)
18
19 #define        GREEN_LED_ON                GPIO_clear_pin(GREEN_LED_PORT, GREEN_LED_PIN)
20 #define        GREEN_LED_OFF                GPIO_set_pin(GREEN_LED_PORT, GREEN_LED_PIN)
21
22 #define        BLUE_LED_ON                        GPIO_clear_pin(BLUE_LED_PORT, BLUE_LED_PI
23 #define        BLUE_LED_OFF                GPIO_set_pin(BLUE_LED_PORT, BLUE_LED_PIN)
24
25 static void tick_int(void);
26
27 int main(void) {
28         GPIO_init(1);
29
30         GPIO_set_dir(RED_LED_PORT,
31                                 RED_LED_PIN,
32                                 GPIO_DIR_OUTPUT,
33                                 LED_OFF_STATE);
34
35         GPIO_set_dir(GREEN_LED_PORT,
36                                 GREEN_LED_PIN,
37                                 GPIO_DIR_OUTPUT,
38                                 LED_OFF_STATE);
39
40         GPIO_set_dir(BLUE_LED_PORT,
```

```
41                                      BLUE_LED_PIN,
42                                      GPIO_DIR_OUTPUT,
43                                      LED_OFF_STATE);
44
45          SYSTICK_init(1000, tick_int);
46
47          while(1)
48          {
49
50          }
51
52      return 0 ;
53  }
54
55  static void tick_int(void)
56  {
57          static uint32_t contador_msegs = 0;
58          static uint32_t contador_etapa = 0;
59
60          contador_msegs++;
61          contador_msegs %= 1000;
62
63          if(!contador_msegs)
64          {
65                  switch(contador_etapa)
66                  {
67                  case 0:
68                          GREEN_LED_OFF;
69                          BLUE_LED_OFF;
70                          RED_LED_ON;
71                          break;
72                  case 1:
73                          RED_LED_OFF;
74                          BLUE_LED_OFF;
75                          GREEN_LED_ON;
76                          break;
77                  case 2:
78                          RED_LED_OFF;
79                          GREEN_LED_OFF;
80                          BLUE_LED_ON;
81                          break;
82                  default:
83                          break;
84                  }
85
86                  contador_etapa++;
87                  contador_etapa %= 3;
88          }
89  }
```