# Freescale Kinetis SDK Preview

Hey, everybody, it is about good news. Freescale has already started to develop complete Driver library and middleware for ARM Cortex M0+/M4 Kinetis Families MCU. It is called Kinetis SDK (Software Development Kit), which is similar with STM32 Peripheral Driver Lib and NXP LPCOpen. SDK's goal is to support all Kinetis chip by providing complete IP register access abstract layer (HAL) and complete device driver based on the HAL. User can use the set of driver library conveniently to implement own applications, without reading complicated chip's reference manual. At the same time, the unified driver interface also ensures application's porting between different Kinetis chip easily. In addition, the SDK will provide RTOS support (including MQX, FreeRTOS, and uCOS), USB and TCP/IP protocol stack, detailed application demo as well as various convenient development tools. Now Kinetis SDK is under developing, several versions will be released this year. Coming soon~

Now let's review the API and it's usage of Kinetis SDK I2C, UART, and PIT peripheral drivers. Wish your valuable opinions, and support Freescale to provide better software solutions for our customers. Any of your 'Like', comments and criticism is very precious to us.

# I2C Driver API

Kinetis I2C (Inter-Integrated Circuit) module, which implement both Master function and Slave function.

## I2C Driver Return Value

```
/*! @brief I2C status return codes.*/
typedef enum _i2c_status
{
    kStatus_I2C_Success = 0,
    kStatus_I2C_OutOfRange,
    kStatus_I2C_InvalidArgument,
    kStatus_I2C_Fail,
    kStatus_I2C_Busy,                /*!< The master is already
                                          performing a transfer.*/
    kStatus_I2C_Timeout,             /*!< The transfer timed out.*/
    kStatus_I2C_ReceivedNak,         /*!< The slave device sent a NAK in
                                          response to a byte.*/
    kStatus_I2C_SlaveTxUnderrun,     /*!< I2C Slave TX Underrun error.*/
    kStatus_I2C_SlaveRxOverrun,      /*!< I2C Slave RX Overrun error.*/
    kStatus_I2C_AribtrationLost,     /*!< I2C Arbitration Lost error.*/
} i2c_status_t;
```

## I2C Master API

### I2C Master Initialize

```
/*!
 * @brief Initialize the I2C master mode driver.
 *
 * @param instance The I2C peripheral instance number.
 * @param master   The pointer to the I2C master driver state structure.
 */
void i2c_master_init(uint32_t instance, i2c_master_t * master);
```

### I2C Master Shutdown

```
/*!
 * @brief Shut down the driver.
 *
 * @param master   The pointer to the I2C master driver state structure.
 */
void i2c_master_shutdown(i2c_master_t * master);
```

### I2C Master Bus Parameter Configure

```
/*!
 * @brief Configure the I2C bus to access a device.
 *
 * @param master   The pointer to the I2C master driver state structure.
 * @param device   The pointer to the I2C device information struct.
 */
i2c_status_t i2c_master_configure_bus(i2c_master_t * master, const
i2c_device_t * device);
```

This function is also called by transfer function i2c_master_transfer() as below. Users can skip i2c_status_t i2c_master_configure_bus() function, and call the transfer function directly.

### I2C Master Sync Transfer

```
/*!
 * @brief Perform a blocking read or write transaction on the I2C bus.
 *
 * @param master                    The pointer to the I2C master driver
                                    state structure.
 * @param device                    The pointer to the I2C device
                                    information struct.
 * @param direction                 The direction of an I2C transfer.
 * @param subaddress                The subaddress for a device if it has.
 * @param subaddressLength          The length of the subaddress.
 * @param data                      The pointer to the data to be
                                    transfered.
 * @param dataLength                The length in bytes of the data to be
                                    transfered.
 * @param actualLengthTransferred   The length in bytes of the data
                                    thansfered.
 * @param timeous_ms                A timeout for the transfer in
                                    microseconds.
 */
i2c_status_t i2c_master_transfer(i2c_master_t * master,
                      const i2c_device_t * device,
                      i2c_direction_t direction,
                      uint32_t subaddress,
                      size_t subaddressLength,
                      uint8_t * data,
                      size_t dataLength,
                      size_t * actualLengthTransferred,
                      uint32_t timeout_ms);
```

## I2C Master Application Example

```
/* eg: Master write a byte to Slave, and then read a byte from Slave */
i2c_master_t master; /* define I2C Master control structure */
uint8_t send_buf[1] = {0xAB};       /* Tx buffer */
uint8_t receive_buf[1] = {0xFF};    /* Rx buffer */
/*set I2C Slave device and bus speed */
i2c_device_t slave ={
    .address = 0x3A,      /* Slave address 0x3A */
    .baudRate_kbps = 100,  /* I2C bus frequence 100KHz */
};
void main()
{
    i2c_status_t ret;  /* init I2C0 */
    i2c_master_init(0, &master);
    printf("I2C Initialized!\r\n");
    /* tx "0xAB" to I2C Slave assigned, no subaddress, max time:200ms*/
    ret = i2c_master_transfer(&master, &slave,kI2CWrite, 0, 0,
                              send_buf, sizeof(send_buf), NULL, 200);
    if (ret)
    {
        printf("I2C Master Transfer Failed!code:%d\r\n", ret);
    }
    /* I2C Master read data from Slave*/
    i2c_master_transfer(&master, &slave, kI2CRead, 0, 0, receive_buf,
                        sizeof(receive_buf), NULL, 200);
    if (ret)
    {
        printf("I2C Master Transfer Failed!code:%d\r\n", ret);
    }
    else
    {
        printf("received:%d\r\n", receive_buf[0]);
    }
    I2c_master_shutdown(&master);
}
```

## I2C Slave API

### I2C Slave Initialize

```
/*!
 * @brief Initializes the I2C module.
 *
 * Saves the application callback info, turns on the clock to the module,
 * Enables the device and enables interrupts. Set the I2C to slave mode.
 * IOMux is expected to already be handled in init_hardware().
 *
 * @param instance   Instance number of the I2C module.
 * @param appInfo    Pointer of application information.
 */
void i2c_slave_init(uint32_t instance, i2c_slave_info_t * appInfo);
```

### I2C Slave Shutdown

```
/*!
 * @brief Shuts down the I2C slave driver.
 *
 * Clears the control register and turns off the clock to the module.
 *
 * @param instance  Instance number of the I2C module.
 */
void i2c_slave_shutdown(uint32_t instance);
```

## I2C Slave Application Example

```
/* u8SinkData: Slave receive data from Master */
uint8_t u8SinkData  = 0x00;
/* u8SourceData: Slave transmit data to Master */
uint8_t u8SourceData= 0xCD;
/* callback function that Slave receive data from Master */
static i2c_status_t data_sink(uint8_t sinkByte)
{
    u8SinkData = sinkByte; /* sinkByte is the data received */
    /* return kStatus_I2C_Success means that data is processed successfully */
    return kStatus_I2C_Success;
}
/* callback function that Slave transmit data to Master */
static i2c_status_t data_source(uint8_t * sourceByte)
{
    *sourceByte = u8SourceData;
    return kStatus_I2C_Success;
}
/* error handling callback function */
static void on_error(i2c_status_t error)
{
    switch (error)
    {
    case kStatus_I2C_SlaveTxUnderrun:
        printf("I2C Error (0x%X) - Slave TX Underrun.\r\n", error);
        break;
    case kStatus_I2C_SlaveRxOverrun:
        printf("I2C Error (0x%X) - Slave RX Overrun.\r\n", error);
        break;
    case kStatus_I2C_AribtrationLost:
        printf("I2C Error (0x%X) - Arbitration Lost.\r\n", error);
        break;
    default:
        printf("I2C Error (0x%X) - Unknown Error.\r\n", error);
        break;
    }
}
void main()
{
    i2c_slave_info_t slave;
    /* init the i2c slave */
    i2c_slave_init(0, &slave, 0x3A, data_source, data_sink, on_error);
    /* wait for the master send the data */
    while (u8SinkData != 0xAB)
    {
    }
}
```

# UART Driver API

UART (Universal Asynchronous Receiver/Transmitter) is the most common communicate module.

## UART Driver Data Structure

### Device Status Data Structure

```
/*!
 * @brief Runtime state of the UART driver.
 *
 * This struct holds data that are used by the UART peripheral driver to
 * communicate between the transfer function and the interrupt handler.
The
 * interrupt handler also uses this information to keep track of its
progress.
 * The user is only responsible to pass in the memory for this run-time
state structure where the
 * UART driver will take care of filling out the members.
 */
typedef struct UartState {
    uint32_t instance;                  /*!< UART module instance number.
                                         */
    bool isTransmitInProgress;          /*!< True if there is an active
                                             transmit. */
    bool isTransmitAsync;               /*!< Whether the transmit is
                                             asynchronous or not. */
    const uint8_t * sendBuffer;         /*!< The buffer of data being
                                             sent.*/
    size_t remainingSendByteCount;      /*!< The remaining number of
                                             bytes to be transmitted. */
    size_t transmittedByteCount;        /*!< Number of bytes transmitted
                                             so far. */
    bool isReceiveInProgress;           /*!< True if there is an active
                                             receive. */
    bool isReceiveAsync;                /*!< Whether the receive is
                                             asynchronous or not. */
    uint8_t * receiveBuffer;            /*!< The buffer of received data.
                                         */
    size_t remainingReceiveByteCount;   /*!< The remaining number of
                                             bytes to be received. */
    size_t receivedByteCount;           /*!< Number of bytes received so
                                             far. */
    sync_object_t txIrqSync;            /*!< Used to wait for ISR to
                                             complete its TX business. */
    sync_object_t rxIrqSync;            /*!< Used to wait for ISR to
                                             complete its RX business. */
    uint8_t txFifoEntryCount;           /*!< Number of data word entries
                                             in TX FIFO. */
} uart_state_t;
```

## User Configure Data Structure

```
/*!
 * @brief User configuration structure for UART driver.
 *
 * Use an instance of this struct with uart_init(). This allows you to
configure the
 * most common settings of the UART peripheral with a single function
call. Settings include:
 * UART baud rate; UART parity mode: disabled (default), or even or odd;
the number of stop bits;
 * the number of bits per data word.
 */
typedef struct UartUserConfig {
    uint32_t baudRate;                  /*!< UART baud rate*/
    uart_parity_mode_t parityMode;      /*!< parity mode, disabled
                                             (default), even, odd */
    uart_stop_bit_count_t stopBitCount; /*!< number of stop bits, 1 stop
                                             bit (default)
                                             or 2 stop bits */
    uart_bit_count_per_char_t bitCountPerChar; /*!< number of bits, 8-bit
                                                    (default) or 9-bit in
                                                    a word (up to 10-bits
                                                    in some UART
                                                    instances) */
} uart_user_config_t;
```

## UART API Function

### UART Initialize

```
/*!
 * @brief This function initializes a UART instance for operation.
 *
 * This function will initialize the run-time state structure to keep
track of the on-going
 * transfers, ungate the clock to the UART module, initialize the module
 * to user defined settings and default settings, configure the IRQ state
structure and enable
 * the module-level interrupt to the core, and enable the UART module
transmitter and receiver.
 * The following is an example of how to set up the uart_state_t and the
 * uart_user_config_t parameters and how to call the uart_init function
by passing
 * in these parameters:
   @code
    uart_user_config_t uartConfig;
    uartConfig.baudRate = 9600;
    uartConfig.bitCountPerChar = kUart8BitsPerChar;
    uartConfig.parityMode = kUartParityDisabled;
    uartConfig.stopBitCount = kUartOneStopBit;
    uart_state_t uartState;
    uart_init(uartInstance, &uartConfig, &uartState);
    @endcode
 *
 * @param uartInstance The UART module instance number.
 * @param uartUserConfig The user configuration structure of type
uart_user_config_t. The user
 *  is responsbile to fill out the members of this structure and to pass
the pointer of this struct
 *  into this function.
 * @param uartState A pointer to the UART driver state structure memory.
The user is only
 *  responsible to pass in the memory for this run-time state structure
where the UART driver
 *  will take care of filling out the members. This run-time state
structure keeps track of the
 *  current transfer in progress.
 * @return An error code or kStatus_UART_Success.
 */
uart_status_t uart_init(uint32_t uartInstance,
                        const uart_user_config_t * uartUserConfig,
                        uart_state_t * uartState);
```

### UART Shutdown

```
/*!
 * @brief This function shuts down the UART by disabling interrupts and
the transmitter/receiver.
 *
 * This function disables the UART interrupts, disables the transmitter
and receiver, and
 * flushes the FIFOs (for modules that support FIFOs).
 *
 * @param uartState A pointer to the UART driver state structure.
 */
void uart_shutdown(uart_state_t * uartState);
```

## UART Sync Transmit

```
/*!
 * @brief This function sends (transmits) data out through the UART
module using a blocking method.
 *
 * A blocking (also known as synchronous) function means that the
function does not return until
 * the transmit is complete. This blocking function is used to send data
through the UART port.
 *
 * @param uartInstance The UART module instance number.
 * @param sendBuffer A pointer to the source buffer containing 8-bit data
chars to send.
 * @param txByteCount The number of bytes to send.
 * @param timeout A timeout value for RTOS abstraction sync control in
milli-seconds (ms).
 * @return An error code or kStatus_UART_Success.
 */
uart_status_t uart_send_data(uart_state_t * uartState,
                        uint8_t * sendBuffer, uint32_t txByteCount,
                        uint32_t timeout);
```

## UART ASync Transmit

```
/*!
 * @brief This function sends (transmits) data through the UART module
using a non-blocking method.
 *
 * A non-blocking (also known as synchronous) function means that the
function returns
 * immediately after initiating the transmit function. The application
has to get the
 * transmit status to see when the transmit is complete. In other words,
after calling non-blocking
 * (asynchronous) send function, the application must get the transmit
status to check if transmit
 * is completed or not.
 * The asynchronous method of transmitting and receiving allows the UART
to perform a full duplex
 * operation (simultaneously transmit and receive).
 *
 * @param uartState A pointer to the UART driver state structure.
 * @param sendBuffer A pointer to the source buffer containing 8-bit data
chars to send.
 * @param txByteCount The number of bytes to send.
 * @return An error code or kStatus_UART_Success.
 */
uart_status_t uart_send_data_async(uart_state_t * uartState,
                              uint8_t * sendBuffer,
                              uint32_t txByteCount);
```

## UART Send State Obtain

```
/*!
 * @brief This function returns whether the previous UART transmit has
finished.
 *
 * When performing an async transmit, the user can call this function to
ascertain the state of the
 * current transmission: in progress (or busy) or complete (success). In
addition, if the
 * transmission is still in progress, the user can obtain the number of
words that have been
 * currently transferred.
 *
 * @param uartState A pointer to the UART driver state structure.
 * @param bytesTransmitted A pointer to a value that is filled in with
the number of bytes that
 *                          are sent in the active transfer.
 *
 * @retval kStatus_UART_Success The transmit has completed successfully.
 * @retval kStatus_UART_TxBusy The transmit is still in progress. @a
bytesTransmitted is
 *      filled with the number of bytes which are transmitted up to that
point.
 */
uart_status_t uart_get_transmit_status(uart_state_t * uartState, uint32_t
* bytesTransmitted);
```

## UART Sync Receive

```
/*!
 * @brief This function gets (receives) data from the UART module using a
blocking method.
 *
 * A blocking (also known as synchronous) function means that the
function does not return until
 * the receive is complete. This blocking function is used to send data
through the UART port.
 *
 * @param uartState A pointer to the UART driver state structure.
 * @param rxBuffer A pointer to the buffer containing 8-bit read data
chars received.
 * @param requestedByteCount The number of bytes to receive.
 * @param timeout A timeout value for RTOS abstraction sync control in
milli-seconds (ms).
 * @return An error code or kStatus_UART_Success.
 */
uart_status_t uart_receive_data(uart_state_t * uartState,
                          uint8_t * rxBuffer,
                          uint32_t requestedByteCount,
                          uint32_t timeout);
```

### UART ASync Receive

```
/*!
 * @brief This function gets (receives) data from the UART module using a
non-blocking method.
 *
 * A non-blocking (also known as synchronous) function means that the
function returns
 * immediately after initiating the receive function. The application has
to get the
 * receive status to see when the receive is complete. In other words,
after calling non-blocking
 * (asynchronous) get function, the application must get the receive
status to check if receive
 * is completed or not.
 * The asynchronous method of transmitting and receiving allows the UART
to perform a full duplex
 * operation (simultaneously transmit and receive).
 *
 * @param uartState A pointer to the UART driver state structure.
 * @param rxBuffer  A pointer to the buffer containing 8-bit read data
chars received.
 * @param requestedByteCount The number of bytes to receive.
 * @return An error code or kStatus_UART_Success.
 */
uart_status_t uart_receive_data_async(uart_state_t * uartState,
                                      uint8_t * rxBuffer,
                                      uint32_t requestedByteCount);
```

### UART Receive State Obtain

```
/*!
 * @brief This function returns whether the previous UART receive is
complete.
 *
 * When performing an async receive, the user can call this function to
ascertain the state of the
 * current receive progress: in progress (or busy) or complete (success).
In addition, if the
 * receive is still in progress, the user can obtain the number of words
that have been
 * currently received.
 *
 * @param uartState A pointer to the UART driver state structure.
 * @param bytesReceived A pointer to a value that is filled in with the
number of bytes which
 *                      are received in the active transfer.
 *
 * @retval kStatus_UART_Success The receive has completed successfully.
 * @retval kStatus_UART_RxBusy The receive is still in progress. @a
bytesReceived is
 *     filled with the number of bytes which are received up to that
point.
 */
uart_status_t uart_get_receive_status(uart_state_t * uartState, uint32_t
* bytesReceived);
```

### UART ASync Transmission Terminate

```
/*!
 * @brief This function terminates an asynchronous UART transmission
early.
 *
 * During an async UART tranmission, the user has the option to terminate
the transmission early
 * if the transmission is still in progress.
 *
 * @param uartState A pointer to the UART driver state structure.
 *
 * @retval kStatus_UART_Success The transmit was succesful.
 * @retval kStatus_UART_NoTransmitInProgress No transmission is currently
in progress.
 */
uart_status_t uart_abort_sending_data(uart_state_t * uartState);
```

### UART ASync Receive Terminate

```
/*!
 * @brief This function terminates an asynchronous UART receive early.
 *
 * During an async UART receive, the user has the option to terminate the
receive early
 * if the receive is still in progress.
 *
 * @param uartState A pointer to the UART driver state structure.
 *
 * @retval kStatus_UART_Success The receive was succesful.
 * @retval kStatus_UART_NoTransmitInProgress No receive is currently in
progress.
 */
uart_status_t uart_abort_receiving_data(uart_state_t * uartState);
```

## UART Application Example (Asynchronous)

```
void main()
{
    uint8_t txBuff[]   = "123456789";
    uint8_t rxBuff[10] = {0};
    uint32_t bytesTransmittedCount = 0;
    uint32_t bytesReceivedCount = 0;  /* 初始化串口 */
    uart_user_config_t uartConfig;
    uart_state_t uartState;
    uartConfig.baudRate = 9600;               /* baud rate */
    uartConfig.bitCountPerChar = kUart8BitsPerChar; /* 8bit */
    uartConfig.parityMode = kUartParityDisabled;/* no parity */
    uartConfig.stopBitCount = kUartOneStopBit;   /* 1 stop bit*/
    uart_init(0, &uartConfig, &uartState); /* init UART0 */
    /* start up asynchronous transmission */
    uart_send_data_async(&uartState, txBuff,strlen((const char*)txBuff
    While (1)
    {
        if (uart_get_transmit_status(&uartState, &bytesTransmittedCount)
              == kStatus_UART_TxBusy)
        {
            If (bytesTransmittedCount == 3)
            {
                uart_abort_sending_data(&uartState);
                break;
            }
        }
    }
    /* start up synchronous reception */
    uart_receive_data_async(&uartState, rxBuff, sizeof(rxBuff));
    /* stop receiving after receiving 3 byte */
    while(1)
    {
        if (uart_get_receive_status(&uartState, &bytesReceivedCount)
             == kStatus_UART_RxBusy)
        {
            if(bytesReceivedCount == 3)
            {
                uart_abort_receiving_data(&uartState);
                break;
            }
        }
    }
}
```

# PIT Driver API

PIT(Periodic Interrupt Timer) is the common timer module in Kinetis chips. PIT is used to achieve timer function mainly for user, and as periodic hardware trigger for ADC , DMA.

## PIT API Function

### PIT Module Initialize

```
/*!
 * @brief Initialize PIT module.
 *
 * This function must be called before calling all the other PIT driver
functions.
 * This function un-gates the PIT clock and enables the PIT module. The
isRunInDebug
 * passed into function will affect all timer channels.
 *
 * @param isRunInDebug Timers run or stop in debug mode.
 *        - true:  Timers continue to run in debug mode.
 *        - false: Timers stop in debug mode.
 */
void pit_init_module(bool isRunInDebug);
```

### PIT Module Shutdown

```
/*!
 * @brief Disable PIT module and gate control.
 *
 * This function disables all PIT interrupts and PIT clock. It then gates
the
 * PIT clock control. pit_init_module must be called if you want to use
PIT again.
 */
void pit_shutdown(void);
```

## PIT Channel Initialize

```
/*!
 * @brief Initialize PIT channel.
 *
 * This function initialize PIT timers by channel. Pass in timer number
and its
 * config structure. Timers do not start counting by default after
calling this
 * function. Function pit_timer_start must be called to start timer
counting.
 * Call pit_set_timer_period_us to re-set the period.
 *
 * Here is an example demonstrating how to define a PIT channel config
structure:
   @code
   pit_config_t pitTestInit = {
        .isInterruptEnabled = true,
        // Only takes effect when chain feature is available.
        // Otherwise, pass in arbitrary value(true/false).
        .isTimerChained = false,
        // In unit of microseconds.
        .periodUs = 1000,
   };
   @endcode
 *
 * @param timer Timer channel number.
 * @param config PIT channel configuration structure.
 */
void pit_init_channel(uint32_t timer, const pit_config_t * config);
```

## PIT Timer Start

```
/*!
 * @brief Start timer counting.
 *
 * After calling this function, timers load period value, count down to 0
and
 * then load the respective start value again. Each time a timer reaches
0,
 * it generates a trigger pulse and sets the timeout interrupt flag.
 *
 * @param timer Timer channel number.
 */
void pit_timer_start(uint32_t timer);
```

### PIT Timer Stop

```
/*!
 * @brief Stop timer counting.
 *
 * This function stops every timer counting. Timers reload their periods
 * respectively after the next time they call pit_timer_start.
 *
 * @param timer Timer channel number.
 */
void pit_timer_stop(uint32_t timer);
```

### PIT Timer Period Configure

```
/*!
 * @brief Set timer period in microsecond units.
 *
 * The period range depends on the frequency of PIT source clock. If the
required period
 * is out of range, use the lifetime timer, if applicable.
 *
 * @param timer Timer channel number.
 * @param us Timer period in microseconds.
 */
void pit_set_timer_period_us(uint32_t timer, uint32_t us);
```

### Current Timer Value Read

```
/*!
 * @brief Read current timer value in microsecond units.
 *
 * This function returns an absolute time stamp in microsecond units.
 * One common use of this function is to measure the running time of a
part of
 * code. Call this function at both the beginning and end of code; the
time
 * difference between these two time stamps is the running time (Make
sure the
 * running time will not exceed the timer period). The time stamp
returned is
 * up-counting.
 *
 * @param timer Timer channel number.
 * @return Current timer value in microseconds.
 */
uint32_t pit_read_timer_us(uint32_t timer);
```

### Lifetime Timer Period Configure

```
/*!
 * @brief Set lifetime timer period.
 *
 * Timer 1 must be chained with timer 0 before using the lifetime timer.
The period
 * range is restricted by "period * pitSourceClock < max of an uint64_t
integer",
 * or it may cause an overflow and be unable to set the correct period.
 *
 * @param period Lifetime timer period in microseconds.
 */
void pit_set_lifetime_timer_period_us(uint64_t us);
```

### Current Lifetime Value Read

```
/*!
 * @brief Read current lifetime value in microseconds.
 *
 * This feature returns an absolute time stamp in microsecond units. The
time stamp
 * value will not exceed the timer period. The timer is up-counting.
 *
 * @return Current lifetime timer value in microseconds.
 */
uint64_t pit_read_lifetime_timer_us(void);
```

### PIT ISR Callback Register

```
/*!
 * @brief Register pit isr callback function.
 *
 * System default ISR interfaces are already defined in fsl_pit_irq.c.
Users
 * can either edit these ISRs or use this function to register a callback
 * function. The default ISR runs the callback function if there is one
 * installed.
 *
 * @param timer    Timer channel number.
 * @param function Pointer to pit isr callback function.
 */
void pit_register_isr_callback_function(uint32_t timer,
pit_isr_callback_t function);
```

## PIT Application Example

### Single channel timer

Single channel timer can define a 32 bit counter.

```c
/* PIT interrupt callback function */
void pit_isr_callback()
{
    /* toggle LED GPIO pin, light on or light off */
    gpio_toggle_pin_output(LEDPIN);
}

void main()
{
    /* timer period is 4000us */
    uint32_t timerPeriod = 4000;

    /* PIT channel 0 ,init structure */
    pit_config_t pitTestInit = {
        .isInterruptEnabled = true,
        .isTimerChained = false,
        .periodUs = timerPeriod,
    };

    /* init PIT module */
    pit_init_module(true);

    /* init PIT channel 0 */
    pit_init_channel(0, &pitTestInit);

    pit_register_isr_callback_function(0,
                           pit_isr_callback);

    /* set timer period */
    pit_set_timer_period_us(0, timerPeriod);
    /* start the timer */
    pit_timer_start(0);

    while (1)
    {

    }
}
```

## Lifetimer

Lifetimer merges the two channels of 32 bit counter into a 64 bit counter, in order to realize the configuration function of 64 bit timer period. Lifetimer can achieve quit a long time period of time function.

```
/* PIT interrupt callback function */
void pit_isr_callback()
{
   /* toggle LED GPIO pin, light on or light off */
    gpio_toggle_pin_output(LEDPIN);
}
void main()
{
    /* timer period */
    uint64_t lifeTimePeriod = 100000;
    /* PIT channel 0 ,init structure */
    pit_config_t pit0TestInit = {
        .isInterruptEnabled = false,
        .isTimerChained = false,
        .periodUs = 0,
    };
    /* PIT channel 1 ,init structure, enable 0,1 channel */
    pit_config_t pit1TestInit = {
        .isInterruptEnabled = true,
        .isTimerChained = true,
        .periodUs = 0,
    };
    /* init PIT module */
    pit_init_module(true);
    /* init PIT channel 0 an channel 1 */
    pit_init_channel(0, &pit0TestInit);
    pit_init_channel(1, &pit1TestInit);
    /* register callback function */
    pit_register_isr_callback_function(1, pit_isr_callback);
    /* set lifetime timer period */
    pit_set_lifetime_timer_period_us(lifeTimePeriod);
    /* start timer */
    pit_timer_start(0);
    pit_timer_start(1);

    while (1){ }
}
```

Thanks for your time, is the API ok?

Don’t forget to put your precious comments~