F2806x Peripheral Driver Library

USER'S GUIDE

Copyright

Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
http://www.ti.com/c2000

Revision Information

This is version 2.07.00.00 of this document, last updated on Fri Nov 17 18:41:59 IST 2023.

Table of Contents

1 Introduction

 The Texas Instruments® C2000Ware® Peripheral Driver Library is a set of drivers for accessing a subset of the peripherals found on the f2806x family of C2000 microcontrollers. While they are not drivers in the pure operating system sense (that is, they do not have a common interface and do not connect into a global device driver infrastructure), they do provide a mechanism that makes it easy to use the device's peripherals.

The capabilities and organization of the drivers are governed by the following design goals:

They are written entirely in C except where absolutely not possible.

They demonstrate how to use the peripheral in its common mode of operation.

They are easy to understand.

They are reasonably efficient in terms of memory and processor usage.

They are as self-contained as possible.

Where possible, computations that can be performed at compile time are done there instead of at run time.

Some consequences of these design goals are:

The drivers are not necessarily as efficient as they could be (from a code size and/or execution speed point of view). While the most efficient piece of code for operating a peripheral would be written in assembly and custom tailored to the specific requirements of the application, further size optimizations of the drivers would make them more difficult to understand.

The drivers do not support the full capabilities of the hardware. Some of the peripherals provide complex capabilities which cannot be utilized by the drivers in this library, though the existing code can be used as a reference upon which to add support for the additional capabilities.

The APIs have a means of removing all error checking code. Because the error checking is usually only useful during initial program development, it can be removed to improve code size and speed.

For many applications, the drivers can be used as is. But in some cases, the drivers will have to be enhanced or rewritten in order to meet the functionality, memory, or processing requirements of the application. If so, the existing driver can be used as a reference on how to operate the peripheral.

Source Code Overview

The following is an overview of the organization of the peripheral driver library source code.

`EULA.txt`	The full text of the End User License Agreement that covers the use of this software package.
`driverlib/`	This directory contains the source code for the drivers.

`hw_*.h`	Header files, one per peripheral, that describe all the registers and the bit fields within those registers for each peripheral. These header files are used by the drivers to directly access a peripheral, and can be used by application code to bypass the peripheral driver library API.
`inc/`	This directory holds the part specific header files used for the direct register access programming model.

Because C2000Ware still provides the same support for the f2806x devices that C2000 users are acustomed to, the driverlib package developed for the f2806x devices contains only a minimal set of peripheral drivers. This driverlib implementation exists only to support the USB stack and its associated example applications. Customers are free to reuse all software contained in this library, but there are currently no plans to expand this library to support the full Stellaris® API.

2 Programming Model

 The peripheral driver library provides support for two programming models: the direct register access model and the software driver model. Each model can be used independently or combined, based on the needs of the application or the programming environment desired by the developer.

Each programming model has advantages and disadvantages. Use of the direct register access model generally results in smaller and more efficient code than using the software driver model. However, the direct register access model requires detailed knowledge of the operation of each register and bit field, as well as their interactions and any sequencing required for proper operation of the peripheral; the developer is insulated from these details by the software driver model, generally requiring less time to develop applications. In the direct register access model, the peripherals are programmed by the application by writing values directly into the peripheral's registers. A set of macros is provided that simplifies this process. These macros are stored in contained in the `inc/hw_types.h` file. By also including the header file that matches the peripheral being used and the `inc/hw_memmap.h` file, macros are available for accessing all registers for that given peripheral, as well as all bit fields within those registers.

The defines used by the direct register access model follow a naming convention that makes it easier to know how to use a particular macro. The rules are as follows:

Values that end in `_R` are used to access the value of a register. For example, `SSI0_CR0_R` is used to access the `CR0` register in the SSI0 module.

Values that end in `_M` represent the mask for a multi-bit field in a register. If the value placed in the multi-bit field is a number, there is a macro with the same base name but ending with `_S` (for example, `SSI_CR0_SCR_M` and `SSI_CR0_SCR_S`). If the value placed into the multi-bit field is an enumeration, then there are a set of macros with the same base name but ending with identifiers for the various enumeration values (for example, the `SSI_CR0_FRF_M` macro defines the bit field, and the `SSI_CR0_FRF_NMW`, `SSI_CR0_FRF_TI`, and `SSI_CR0_FRF_MOTO` macros provide the enumerations for the bit field).

Values that end in `_S` represent the number of bits to shift a value in order to align it with a multi-bit field. These values match the macro with the same base name but ending with `_M`.

All other macros represent the value of a bit field.

All register name macros start with the module name and instance number (for example, `SSI0` for the first SSI module) and are followed by the name of the register as it appears in the data sheet (for example, the `CR0` register in the data sheet results in `SSI0_CR0_R`).

All register bit fields start with the module name, followed by the register name, and then followed by the bit field name as it appears in the data sheet. For example, the `SCR` bit field in the `CR0` register in the `SSI` module will be identified by `SSI_CR0_SCR...`. In the case where the bit field is a single bit, there will be nothing further (for example, `SSI_CR0_SPH` is a single bit in the `CR0` register). If the bit field is more than a single bit, there will be a mask value (`_M`) and either a shift (`_S`) if the bit field contains a number or a set of enumerations if not.

Given these definitions, the `CR0` register can be programmed as follows:

$$\text{SSI0}_{CR0_R} = ((5 << SSI_{CR0_{SCR_S}})|SSI_{CR0_{SPH}}|SSI_{CR0_{SPO}}|SSI_{CR0_{FRF_{MOTO}}}|SSI_{CR0_{DSS_8}});$$

Alternatively, the following has the same effect (although it is not as easy to understand):

SSI0$_{C}R0_{R} = 0x000005c7$;

Extracting the value of the `SCR` field from the `CR0` register is as follows:

ulValue = (SSI0$_{C}R0_{R}SSI_{C}R0_{S}CR_{M}) >> SSI0_{C}R0_{S}CR_{S}$;

In the software driver model, the API provided by the peripheral driver library is used by applications to control the peripherals. Because these drivers provide complete control of the peripherals in their normal mode of operation, it is possible to write an entire application without direct access to the hardware. This method provides for rapid development of the application without requiring detailed knowledge of how to program the peripherals.

Corresponding to the direct register access model example, the following call also programs the `CR0` register in the SSI module (though the register name is hidden by the API):

SSIConfigSetExpClk(SSI0$_{B}ASE, 50000000, SSI_{F}RF_{M}OTO_{M}ODE_{3}, SSI_{M}ODE_{M}ASTER, 1000000, 8)$;

The resulting value in the `CR0` register might not be exactly the same because SSIConfigSetExpClk() may compute a different value for the `SCR` bit field than what was used in the direct register access model example.

The drivers in the peripheral driver library are described in the remaining chapters in this document. They combine to form the software driver model. The direct register access model and software driver model can be used together in a single application, allowing the most appropriate model to be applied as needed to any particular situation within the application. For example, the software driver model can be used to configure the peripherals (because this is not performance critical) and the direct register access model can be used for operation of the peripheral (which may be more performance critical). Or, the software driver model can be used for peripherals that are not performance critical (such as a UART used for data logging) and the direct register access model for performance critical peripherals.

3 Interrupt Controller (PIE)

 The interrupt controller API provides a set of functions for dealing with the Peripheral Interrupt Expansion Controller (PIE). Functions are provided to enable and disable interrupts, and register interrupt handlers.

The PIE provides global interrupt maskin, prioritization, and handler dispatching. Individual interrupt sources can be masked, and the processor interrupt can be globally masked as well (without affecting the individual source masks).

The PIE is tightly coupled with the C28x microprocessor. When the processor responds to an interrupt, PIE will supply the address of the function to handle the interrupt directly to the processor. This eliminates the need for a global interrupt handler that queries the interrupt controller to determine the cause of the interrupt and branch to the appropriate handler, reducing interrupt response time.

Interrupt handlers can be configured in two ways; statically at compile time and dynamically at run time. Static configuration of interrupt handlers is accomplished by editing the interrupt handler table in the application's startup code. When statically configured, the interrupts must be explicitly enabled in PIE via IntEnable() before the processor will respond to the interrupt (in addition to any interrupt enabling required within the peripheral itself). Alternatively, interrupts can be configured at run-time using IntRegister(). When using IntRegister(), the interrupt must also be enabled as before.

Correct operation of the PIE controller requires that the vector table be placed at 0xD000 in RAM. Failure to do so will result in an incorrect vector address being fetched in response to an interrupt. The vector table is in a section called "PieVectTableFile" and should be placed appropriately with a linker script.

This driver is contained in `driverlib/interrupt.c`, with `driverlib/interrupt.h` containing the API definitions for use by applications.

3.1 API Functions

Functions

void IntDisable (unsigned long ulInterrupt)

void IntEnable (unsigned long ulInterrupt)

void Interrupt_disable (uint32_t interruptNumber)

void Interrupt_enable (uint32_t interruptNumber)

tBoolean IntMasterDisable (void)

tBoolean IntMasterEnable (void)

void IntRegister (unsigned long ulInterrupt, void (*pfnHandler)(void))

void IntUnregister (unsigned long ulInterrupt)

3.1.1 Detailed Description

The primary function of the interrupt controller API is to manage the interrupt vector table used by the PIE to dispatch interrupt requests. Registering an interrupt handler is a simple matter of inserting the handler address into the table. By default, the table is filled with pointers to an internal handler that loops forever; it is an error for an interrupt to occur when there is no interrupt handler registered to process it. Therefore, interrupt sources should not be enabled before a handler has been registered, and interrupt sources should be disabled before a handler is unregistered. Interrupt handlers are managed with IntRegister() and IntUnregister().

Each interrupt source can be individually enabled and disabled via IntEnable() and IntDisable(). The processor interrupt can be enabled and disabled via IntMasterEnable() and IntMasterDisable(); this does not affect the individual interrupt enable states. Masking of the processor interrupt can be utilized as a simple critical section (only NMI will interrupt the processor while the processor interrupt is disabled), though this will have adverse effects on the interrupt response time.

3.1.2 Function Documentation

3.1.2.1

unsigned long ulInterrupt) Disables an interrupt.

Parameters *ulInterrupt* specifies the interrupt to be disabled.

The specified interrupt is disabled in the interrupt controller. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Prototype:

```
IntDisable void IntDisable(
```

Returns None.

Referenced by SysTickIntDisable(), SysTickIntUnregister(), UARTRXIntUnregister(), UARTTXIntUnregister(), and USBIntUnregister().

3.1.2.2 void IntEnable (

unsigned long ulInterrupt)

Enables an interrupt.

Parameters *ulInterrupt* specifies the interrupt to be enabled.

The specified interrupt is enabled in the interrupt controller. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Returns None.

Referenced by SysTickIntEnable(), SysTickIntRegister(), UARTRXIntRegister(), UARTTXIntRegister(), and USBIntRegister().

3.1.2.3 void Interrupt_disable (uint32_t interruptNumber)

Interrupt_disable

Parameters *interruptNumber* is the interrupt number.

Disable interrupts in the proper way. Written like the Potenza and Soprano interrupt driver.

Returns None.

References IntMasterDisable(), and IntMasterEnable().

3.1.2.4 void Interrupt_enable (uint32_t interruptNumber)

Interrupt_enable

Parameters *interruptNumber* is the interrupt number.

Enable interrupts in the proper way. Written like the Potenza and Soprano interrupt driver.

Returns None.

References IntMasterDisable(), and IntMasterEnable().

3.1.2.5 tBoolean IntMasterDisable ( )

Disables the processor interrupt.

Prevents the processor from receiving interrupts. This does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Note Previously, this function had no return value. As such, it was possible to include `interrupt.h` and call this function without having included `hw_types.h`. Now that the return is a `tBoolean`, a compiler error will occur in this case. The solution is to include `hw_types.h` before including `interrupt.h`. Returns Returns **true** if interrupts were already disabled when the function was called or **false** if they were initially enabled.

Referenced by Interrupt_disable(), and Interrupt_enable().

3.1.2.6 tBoolean IntMasterEnable ( )

Enables the processor interrupt.

Allows the processor to respond to interrupts. This does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Note Previously, this function had no return value. As such, it was possible to include `interrupt.h` and call this function without having included `hw_types.h`. Now that the return is a `tBoolean`, a compiler error will occur in this case. The solution is to include `hw_types.h` before including `interrupt.h`. Returns Returns **true** if interrupts were disabled when the function was called or **false** if they were initially enabled.

Referenced by Interrupt_disable(), and Interrupt_enable().

3.1.2.7 void IntRegister (

void unsigned long ulInterrupt,

void(*)(void) pfnHandler)

Registers a function to be called when an interrupt occurs.

Assumes PIE is enabled

Parameters *ulInterrupt* specifies the interrupt in question.

---

pfnHandler is a pointer to the function to be called.

This function is used to specify the handler function to be called when the given interrupt is asserted to the processor. When the interrupt occurs, if it is enabled (via IntEnable()), the handler function will be called in interrupt context. Since the handler function can preempt other code, care must be taken to protect memory or peripherals that are accessed by the handler and other non-handler code.

Note The use of this function (directly or indirectly via a peripheral driver interrupt register function) moves the interrupt vector table from flash to SRAM. Therefore, care must be taken when linking the application to ensure that the SRAM vector table is located at the beginning of SRAM; otherwise NVIC will not look in the correct portion of memory for the vector table (it requires the vector table be on a 1 kB memory alignment). Normally, the SRAM vector table is so placed via the use of linker scripts; some tool chains, such as the evaluation version of RV-MDK, do not support linker scripts and therefore will not produce a valid executable. See the discussion of compile-time versus run-time interrupt handler registration in the introduction to this chapter. Returns None.

Referenced by SysTickIntRegister(), UARTRXIntRegister(), UARTTXIntRegister(), and USBIntRegister().

3.1.2.8 void IntUnregister (

unsigned long ulInterrupt)

Unregisters the function to be called when an interrupt occurs.

Parameters *ulInterrupt* specifies the interrupt in question.

This function is used to indicate that no handler should be called when the given interrupt is asserted to the processor. The interrupt source will be automatically disabled (via IntDisable()) if necessary.

See Also IntRegister() for important information about registering interrupt handlers. Returns None.

Referenced by SysTickIntUnregister(), UARTRXIntUnregister(), UARTTXIntUnregister(), and USBIntUnregister().

The following example shows how to use the Interrupt Controller API to register an interrupt handler and enable the interrupt.

// // The interrupt handler function. // extern void IntHandler(void);

// // Register the interrupt handler function for interrupt 5. // IntRegister(5, IntHandler);

// // Enable interrupt 5. // IntEnable(5);

// // Enable interrupt 5. // IntMasterEnable();

4 System Control

?? System control determines the overall operation of the device. It controls the clocking of the device, the set of peripherals that are enabled, and configuration of the device and its resets.

The device can be clocked from one of three sources: an external oscillator, the internal oscillator, or an external single ended clock source. The PLL can used with any of the three oscillators as its input. Be careful to ensure that the selected PLL frequency is never greater than 120 MHz.

The whole device can operate from the same clock, however certain applications will want to make use of additional clocking options within the device. For instance the USB peripheral requires a 60 MHz clock to operate correctly. To generate this clock the second on chip PLL can be used.

Please keep in mind the system control drivers only offer the limited functionality needed to support the USB protocol stack and example applications.

This driver is contained in `driverlib/sysctl.c`, with `driverlib/sysctl.h` containing the API definitions for use by applications.

4.1 API Functions

Macros

#define SYSTEM_CLOCK_SPEED

Functions

unsigned long SysCtlClockGet (unsigned long u32ClockIn)

void SysCtlDelay (unsigned long ulCount)

void SysCtlPeripheralDisable (unsigned long ulPeripheral)

void SysCtlPeripheralEnable (unsigned long ulPeripheral)

tBoolean SysCtlPeripheralPresent (unsigned long ulPeripheral)

void SysCtlPeripheralReset (unsigned long ulPeripheral)

void SysCtlUSBPLLDisable (void)

void SysCtlUSBPLLEnable (void)

4.1.1 Detailed Description

The SysCtl API is broken up into three groups of functions: those that provide device information, those that deal with device clocking, and those that provide peripheral control.

Information about the device is provided by SysCtlPeripheralPresent().

Clocking of the device is configured with SysCtlUSBPLLEnable() and SysCtlUSBPLLDisable(). Information about device clocking is provided by SysCtlClockGet().

Peripheral enabling and reset are controlled with SysCtlPeripheralEnable() and SysCtlPeripheralDisable().

4.1.2 Macro Definition Documentation

4.1.2.1

Defined System Clock Speed (CPU speed). Adjust this to reflect your actual clock speed.

Referenced by SysCtlClockGet().

4.1.3 Function Documentation

4.1.3.1

unsigned long u32ClockIn) Gets the processor clock rate.

Parameters *u32ClockIn* is the processor clock

This function determines the clock rate of the processor clock.

Prototype:

```
SysCtlClockGet unsigned long SysCtlClockGet(
```

Note Because of the many different clocking options available, this function cannot determine the clock speed of the processor. This function should be modified to return the actual clock speed of the processor in your specific application. Returns The processor clock rate.

References SYSTEM_CLOCK_SPEED.

4.1.3.2 void SysCtlDelay (

unsigned long ulCount)

Provides a small delay.

Parameters *ulCount* is the number of delay loop iterations to perform.

This function provides a means of generating a constant length delay. It is written in assembly to keep the delay consistent across tool chains, avoiding the need to tune the delay based on the tool chain in use.

The loop takes 3 cycles/loop.

Returns None.

4.1.3.3 void SysCtlPeripheralDisable (unsigned long ulPeripheral)

Disables a peripheral.

Parameters *ulPeripheral* is the peripheral to disable.

Peripherals are disabled with this function. Once disabled, they will not operate or respond to register reads/writes.

The *ulPeripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_SPI0**, **SYSCTL_PERIPH_SPI1**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_DMA**, **SYSCTL_PERIPH_USB0**.

Returns None.

4.1.3.4 void SysCtlPeripheralEnable (unsigned long ulPeripheral)

Enables a peripheral.

Parameters *ulPeripheral* is the peripheral to enable.

Peripherals are enabled with this function. At power-up, all peripherals are disabled; they must be enabled in order to operate or respond to register reads/writes.

The *ulPeripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_SPI0**, **SYSCTL_PERIPH_SPI1**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_DMA**, **SYSCTL_PERIPH_USB0**.

Returns None.

4.1.3.5 tBoolean SysCtlPeripheralPresent (unsigned long ulPeripheral)

Determines if a peripheral is present.

Parameters *ulPeripheral* is the peripheral in question.

Determines if a particular peripheral is present in the device.

The *ulPeripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_SPI0**, **SYSCTL_PERIPH_SPI1**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_DMA**, **SYSCTL_PERIPH_USB0**.

Returns Returns **true** if the specified peripheral is present and **false** if it is not.

4.1.3.6 void SysCtlPeripheralReset (unsigned long ulPeripheral)

Resets a peripheral

Parameters *ulPeripheral* is the peripheral to reset.

The f2806x devices do not have a means of resetting peripherals via via software. This is a dummy function that does nothing.

Returns None.

4.1.3.7 void SysCtlUSBPLLDisable ( )

Powers down the USB PLL.

This function will disable the USB controller's PLL. The USB registers are still accessible, but the physical layer will no longer function.

Returns None.

4.1.3.8 void SysCtlUSBPLLEnable ( )

Powers up the USB PLL.

This function will enable the USB controller's PLL.

Note Because every application is different, the user will likely have to modify this function to ensure the PLL multiplier is set correctly to achieve the 60 MHz required by the USB controller. Returns None. The following example shows how to use the SysCtl API to configure the device for normal operation.

// // Enable the USB controller // SysCtlPeripheralEnable(SYSCTL_PERIPH_USB0);

5 System Tick (SysTick)

 SysTick is a simple timer that makes use of the CPU Timer0 module within the C28x core. Its intended purpose is to provide a periodic interrupt for a RTOS, but it can be used for other simple timing purposes.

This driver is contained in `driverlib/systick.c`, with `driverlib/systick.h` containing the API definitions for use by applications.

5.1 API Functions

Functions

void void SysTickDisable (void)

void SysTickEnable (void)

void SysTickInit (void)

void SysTickIntDisable (void)

void SysTickIntEnable (void)

void SysTickIntRegister (void (*pfnHandler)(void))

void SysTickIntUnregister (void)

unsigned long SysTickPeriodGet (void)

void SysTickPeriodSet (unsigned long ulPeriod)

unsigned long SysTickValueGet (void)

5.1.1 Detailed Description

The SysTick API is fairly simple, like SysTick itself. There are functions for configuring and enabling SysTick (SysTickInit(), SysTickEnable(), SysTickDisable(), SysTickPeriodSet(), SysTickPeriodGet(), and SysTickValueGet()) and functions for dealing with an interrupt handler for SysTick (SysTickIntRegister(), SysTickIntUnregister(), SysTickIntEnable(), and SysTickIntDisable()).

5.1.2 Function Documentation

5.1.2.1

) Disables the SysTick counter.

This will stop the SysTick counter. If an interrupt handler has been registered, it will no longer be called until SysTick is restarted.

Prototype:

void SysTickDisable void SysTickDisable(

Returns None.

5.1.2.2 void SysTickEnable (

)

Enables the SysTick counter.

This will start the SysTick counter. If an interrupt handler has been registered, it will be called when the SysTick counter rolls over.

Note Calling this function will cause the SysTick counter to (re)commence counting from its current value. The counter is not automatically reloaded with the period as specified in a previous call to SysTickPeriodSet(). If an immediate reload is required, the **NVIC_ST_CURRENT** register must be written to force this. Any write to this register clears the SysTick counter to 0 and will cause a reload with the supplied period on the next clock. Returns None.

5.1.2.3 void SysTickInit (

)

Initializes the Timer0 Module to act as a system tick

Returns None.

5.1.2.4 void SysTickIntDisable (

)

Disables the SysTick interrupt.

This function will disable the SysTick interrupt, preventing it from being reflected to the processor.

Returns None.

References IntDisable().

5.1.2.5 void SysTickIntEnable (

)

Enables the SysTick interrupt.

This function will enable the SysTick interrupt, allowing it to be reflected to the processor.

Note The SysTick interrupt handler does not need to clear the SysTick interrupt source as this is done automatically by NVIC when the interrupt handler is called. Returns None.

References IntEnable().

5.1.2.6 void SysTickIntRegister (

void(∗)(void) pfnHandler)

Registers an interrupt handler for the SysTick interrupt.

Parameters *pfnHandler* is a pointer to the function to be called when the SysTick interrupt occurs.

This sets the handler to be called when a SysTick interrupt occurs.

See Also IntRegister() for important information about registering interrupt handlers. Returns None.

References IntEnable(), and IntRegister().

5.1.2.7 void SysTickIntUnregister (

)

Unregisters the interrupt handler for the SysTick interrupt.

This function will clear the handler to be called when a SysTick interrupt occurs.

See Also IntRegister() for important information about registering interrupt handlers. Returns None.

References IntDisable(), and IntUnregister().

5.1.2.8 unsigned long SysTickPeriodGet (

)

Gets the period of the SysTick counter.

This function returns the rate at which the SysTick counter wraps; this equates to the number of processor clocks between interrupts.

Returns Returns the period of the SysTick counter.

5.1.2.9 void SysTickPeriodSet (

void unsigned long ulPeriod)

Sets the period of the SysTick counter.

Parameters *ulPeriod* is the number of clock ticks in each period of the SysTick counter; must be between 1 and 16, 777, 216, inclusive.

This function sets the rate at which the SysTick counter wraps; this equates to the number of processor clocks between interrupts.

Note Calling this function does not cause the SysTick counter to reload immediately. If an immediate reload is required, the **NVIC_ST_CURRENT** register must be written. Any write to this register clears the SysTick counter to 0 and will cause a reload with the *ulPeriod* supplied here on the next clock after the SysTick is enabled. Returns None.

5.1.2.10 unsigned long SysTickValueGet (

)

Gets the current value of the SysTick counter.

This function returns the current value of the SysTick counter; this will be a value between the period - 1 and zero, inclusive.

Returns Returns the current value of the SysTick counter. The following example shows how to use the SysTick API to configure the SysTick counter and read its value.

unsigned long ulValue;

// // Configure and enable the SysTick counter. // SysTickInit(); SysTickPeriodSet(1000); SysTick-Enable();

// // Delay for some time... //

// // Read the current SysTick value. // ulValue = SysTickValueGet();

6 UART

Programming Example **??** The Universal Asynchronous Receiver/Transmitter (UART) API provides a set of functions for using the C2000 SCI modules. Functions are provided to configure and control the SCI modules, to send and receive data, and to manage interrupts for the UART modules.

The SCI performs the functions of parallel-to-serial and serial-to-parallel conversions. It is very similar in functionality to a 16C550 UART, but is not register-compatible.

This driver is contained in `driverlib/uart.c`, with `driverlib/uart.h` containing the API definitions for use by applications.

6.1 API Functions

Functions

void tBoolean UARTBusy (unsigned long ulBase)

long UARTCharGet (unsigned long ulBase)

long UARTCharGetNonBlocking (unsigned long ulBase)

void UARTCharPut (unsigned long ulBase, unsigned char ucData)

tBoolean UARTCharPutNonBlocking (unsigned long ulBase, unsigned char ucData)

tBoolean UARTCharsAvail (unsigned long ulBase)

void UARTConfigGetExpClk (unsigned long ulBase, unsigned long ulUARTClk, unsigned long *pulBaud, unsigned long *pulConfig)

void UARTConfigSetExpClk (unsigned long ulBase, unsigned long ulUARTClk, unsigned long ulBaud, unsigned long ulConfig)

void UARTDisable (unsigned long ulBase)

void UARTEnable (unsigned long ulBase)

void UARTFIFODisable (unsigned long ulBase)

void UARTFIFOEnable (unsigned long ulBase)

void UARTFIFOLevelGet (unsigned long ulBase, unsigned long *pulTxLevel, unsigned long *pulRxLevel)

void UARTFIFOLevelSet (unsigned long ulBase, unsigned long ulTxLevel, unsigned long ulRxLevel)

void UARTIntClear (unsigned long ulBase, unsigned long ulIntFlags)

void UARTIntDisable (unsigned long ulBase, unsigned long ulIntFlags)

void UARTIntEnable (unsigned long ulBase, unsigned long ulIntFlags)

unsigned long UARTIntStatus (unsigned long ulBase, tBoolean bMasked)

unsigned long UARTParityModeGet (unsigned long ulBase)

void UARTParityModeSet (unsigned long ulBase, unsigned long ulParity)

void UARTRxErrorClear (unsigned long ulBase)

unsigned long UARTRxErrorGet (unsigned long ulBase)

void UARTRXIntRegister (unsigned long ulBase, void (*pfnHandler)(void))

void UARTRXIntUnregister (unsigned long ulBase)

tBoolean UARTSpaceAvail (unsigned long ulBase)

unsigned long UARTTxIntModeGet (unsigned long ulBase)

void UARTTxIntModeSet (unsigned long ulBase, unsigned long ulMode)

void UARTTXIntRegister (unsigned long ulBase, void (*pfnHandler)(void))

void UARTTXIntUnregister (unsigned long ulBase)

6.1.1 Detailed Description

The UART API provides the set of functions required to implement an interrupt driven UART driver. These functions may be used to control any of the available SCI ports on a f2806x microcontroller, and can be used with one port without causing conflicts with the other port.

The UART API is broken into three groups of functions: those that deal with configuration and control of the UART modules, those used to send and receive data, and those that deal with interrupt handling.

Configuration and control of the UART are handled by the UARTConfigGetExpClk(), UARTConfigSetExpClk(), UARTDisable(), UARTEnable(), UARTParityModeGet(), and UARTParityModeSet() functions.

Sending and receiving data via the UART is handled by the UARTCharGet(), UARTCharGetNonBlocking(), UARTCharPut(), UARTCharPutNonBlocking(), UARTBreakCtl(), UARTCharsAvail(), and UARTSpaceAvail() functions.

Managing the UART interrupts is handled by the UARTIntClear(), UARTIntDisable(), UARTIntEnable(), UARTIntRegister(), UARTIntStatus(), and UARTIntUnregister() functions.

6.1.2 Function Documentation

6.1.2.1

unsigned long ulBase) Determines whether the UART transmitter is busy or not.

Parameters *ulBase* is the base address of the UART port.

Allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If **false** is returned, the transmit FIFO is empty and all bits of the last transmitted character, including all stop bits, have left the hardware shift register.

Prototype:

```
UARTBusy tBoolean UARTBusy(
```

Returns Returns **true** if the UART is transmitting or **false** if all transmissions are complete.

6.1.2.2 long UARTCharGet (

unsigned long ulBase)

Waits for a character from the specified port.

Parameters *ulBase* is the base address of the UART port.

Gets a character from the receive FIFO for the specified port. If there are no characters available, this function waits until a character is received before returning.

Returns Returns the character read from the specified port, cast as a *long*.

6.1.2.3 long UARTCharGetNonBlocking (

unsigned long ulBase)

Receives a character from the specified port.

Parameters *ulBase* is the base address of the UART port.

Gets a character from the receive FIFO for the specified port.

This function replaces the original UARTCharNonBlockingGet() API and performs the same actions. A macro is provided in `uart.h` to map the original API to this API.

Returns Returns the character read from the specified port, cast as a *long*. A **-1** is returned if there are no characters present in the receive FIFO. The UARTCharsAvail() function should be called before attempting to call this function.

6.1.2.4 void UARTCharPut (

unsigned long ulBase,

unsigned char ucData)

Waits to send a character from the specified port.

Parameters *ulBase* is the base address of the UART port.

---

ucData is the character to be transmitted.

Sends the character *ucData* to the transmit FIFO for the specified port. If there is no space available in the transmit FIFO, this function waits until there is space available before returning.

Returns None.

6.1.2.5 tBoolean UARTCharPutNonBlocking (

unsigned long ulBase,

unsigned char ucData)

Sends a character to the specified port.

Parameters *ulBase* is the base address of the UART port.

---

ucData is the character to be transmitted.

Writes the character *ucData* to the transmit FIFO for the specified port. This function does not block, so if there is no space available, then a **false** is returned, and the application must retry the function later.

This function replaces the original UARTCharNonBlockingPut() API and performs the same actions. A macro is provided in `uart.h` to map the original API to this API.

Returns Returns **true** if the character was successfully placed in the transmit FIFO or **false** if there was no space available in the transmit FIFO.

6.1.2.6 tBoolean UARTCharsAvail (

unsigned long ulBase)

Determines if there are any characters in the receive FIFO.

Parameters *ulBase* is the base address of the UART port.

This function returns a flag indicating whether or not there is data available in the receive FIFO.

Returns Returns **true** if there is data in the receive FIFO or **false** if there is no data in the receive FIFO.

6.1.2.7 void UARTConfigGetExpClk (

unsigned long ulBase,

unsigned long ulUARTClk,

unsigned long * pulBaud,

unsigned long * pulConfig)

Gets the current configuration of a UART.

Parameters *ulBase* is the base address of the UART port.

---

ulUARTClk is the rate of the clock supplied to the UART module.

---

pulBaud is a pointer to storage for the baud rate.

---

pulConfig is a pointer to storage for the data format.

The baud rate and data format for the UART is determined, given an explicitly provided peripheral clock (hence the ExpClk suffix). The returned baud rate is the actual baud rate; it may not be the exact baud rate requested or an "official" baud rate. The data format returned in *pulConfig* is enumerated the same as the *ulConfig* parameter of UARTConfigSetExpClk().

The peripheral clock will be the same as the processor clock. This will be the value returned by SysCtlClockGet(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtlClockGet()).

This function replaces the original UARTConfigGet() API and performs the same actions. A macro is provided in `uart.h` to map the original API to this API.

Returns None.

6.1.2.8 void UARTConfigSetExpClk (

unsigned long ulBase,

unsigned long ulUARTClk,

unsigned long ulBaud,

unsigned long ulConfig)

Sets the configuration of a UART.

Parameters *ulBase* is the base address of the UART port.

---

ulUARTClk is the rate of the clock supplied to the UART module.

---

ulBaud is the desired baud rate.

---

ulConfig is the data format for the port (number of data bits, number of stop bits, and parity).

This function configures the UART for operation in the specified data format. The baud rate is provided in the *ulBaud* parameter and the data format in the *ulConfig* parameter.

The *ulConfig* parameter is the logical OR of three values: the number of data bits, the number of stop bits, and the parity. **UART_CONFIG_WLEN_8**, **UART_CONFIG_WLEN_7**, **UART_CONFIG_WLEN_6**, and **UART_CONFIG_WLEN_5** select from eight to five data bits per byte (respectively). **UART_CONFIG_STOP_ONE** and **UART_CONFIG_STOP_TWO** select

one or two stop bits (respectively). **UART_CONFIG_PAR_NONE**, **UART_CONFIG_PAR_EVEN**, **UART_CONFIG_PAR_ODD**, **UART_CONFIG_PAR_ONE**, and **UART_CONFIG_PAR_ZERO** select the parity mode (no parity bit, even parity bit, odd parity bit, parity bit always one, and parity bit always zero, respectively).

The peripheral clock will be the same as the processor clock. This will be the value returned by SysCtlClockGet(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtlClockGet()).

This function replaces the original UARTConfigSet() API and performs the same actions. A macro is provided in `uart.h` to map the original API to this API.

Returns None.

References UARTDisable(), and UARTEnable().

6.1.2.9 void UARTDisable (unsigned long ulBase)

Disables transmitting and receiving.

Parameters *ulBase* is the base address of the UART port.

Clears the UARTEN, TXE, and RXE bits, then waits for the end of transmission of the current character, and flushes the transmit FIFO.

Returns None.

Referenced by UARTConfigSetExpClk().

6.1.2.10 void UARTEnable (unsigned long ulBase)

Enables transmitting and receiving.

Parameters *ulBase* is the base address of the UART port.

Sets the UARTEN, TXE, and RXE bits, and enables the transmit and receive FIFOs.

Returns None.

Referenced by UARTConfigSetExpClk().

6.1.2.11 void UARTFIFODisable (unsigned long ulBase)

Disables the transmit and receive FIFOs.

Parameters *ulBase* is the base address of the UART port.

This functions disables the transmit and receive FIFOs in the UART.

722.7

ulTxLevel is the transmit FIFO interrupt level, specified as one of **UART_FIFO_TX1_8**, **UART_FIFO_TX2_8**, **UART_FIFO_TX4_8**, **UART_FIFO_TX6_8**, or **UART_FIFO_TX7_8**.

ulRxLevel is the receive FIFO interrupt level, specified as one of **UART_FIFO_RX1_8**, **UART_FIFO_RX2_8**, **UART_FIFO_RX4_8**, **UART_FIFO_RX6_8**, or **UART_FIFO_RX7_8**.

This function sets the FIFO level at which transmit and receive interrupts are generated.

Returns None.

6.1.2.15 void UARTIntClear (

unsigned long ulBase,

unsigned long ulIntFlags)

Clears UART interrupt sources.

Parameters *ulBase* is the base address of the UART port.

ulIntFlags is a bit mask of the interrupt sources to be cleared.

The specified UART interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being recognized again immediately upon exit.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to UARTIntEnable().

Note Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted). Returns None.

6.1.2.16 void UARTIntDisable (

unsigned long ulBase,

unsigned long ulIntFlags)

Disables individual UART interrupt sources.

Parameters *ulBase* is the base address of the UART port.

ulIntFlags is the bit mask of the interrupt sources to be disabled.

Disables the indicated UART interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to UARTIntEnable().

Returns None.

6.1.2.17 void UARTIntEnable (

unsigned long ulBase,

unsigned long ulIntFlags)

Enables individual UART interrupt sources.

Parameters *ulBase* is the base address of the UART port.

---

ulIntFlags is the bit mask of the interrupt sources to be enabled.

Enables the indicated UART interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ulIntFlags* parameter is the logical OR of any of the following:

UART_INT_OE - Overrun Error interrupt

UART_INT_BE - Break Error interrupt

UART_INT_PE - Parity Error interrupt

UART_INT_FE - Framing Error interrupt

UART_INT_RT - Receive Timeout interrupt

UART_INT_TX - Transmit interrupt

UART_INT_RX - Receive interrupt

UART_INT_DSR - DSR interrupt

UART_INT_DCD - DCD interrupt

UART_INT_CTS - CTS interrupt

UART_INT_RI - RI interrupt

Returns None.

6.1.2.18 unsigned long UARTIntStatus (

unsigned long ulBase,

tBoolean bMasked)

Gets the current interrupt status.

Parameters *ulBase* is the base address of the UART port.

---

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

This returns the interrupt status for the specified UART. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns Returns the current interrupt status, enumerated as a bit field of values described in UARTIntEnable().

6.1.2.19 unsigned long UARTParityModeGet (

unsigned long ulBase)

Gets the type of parity currently being used.

Parameters *ulBase* is the base address of the UART port.

This function gets the type of parity used for transmitting data and expected when receiving data.

Returns Returns the current parity settings, specified as one of **UART_CONFIG_PAR_NONE**, **UART_CONFIG_PAR_EVEN**, **UART_CONFIG_PAR_ODD**, **UART_CONFIG_PAR_ONE**, or **UART_CONFIG_PAR_ZERO**.

6.1.2.20 void UARTParityModeSet (

unsigned long ulBase,

unsigned long ulParity)

Sets the type of parity.

Parameters *ulBase* is the base address of the UART port.

---

ulParity specifies the type of parity to use.

Sets the type of parity to use for transmitting and expect when receiving. The *ulParity* parameter must be one of **UART_CONFIG_PAR_NONE**, **UART_CONFIG_PAR_EVEN**, **UART_CONFIG_PAR_ODD**, **UART_CONFIG_PAR_ONE**, or **UART_CONFIG_PAR_ZERO**. The last two allow direct control of the parity bit; it is always either one or zero based on the mode.

Returns None.

6.1.2.21 void UARTRxErrorClear (

unsigned long ulBase)

Clears all reported receiver errors.

Parameters *ulBase* is the base address of the UART port.

This function is used to clear all receiver error conditions reported via UARTRxErrorGet(). If using the overrun, framing error, parity error or break interrupts, this function must be called after clearing the interrupt to ensure that later errors of the same type trigger another interrupt.

Returns None.

6.1.2.22 unsigned long UARTRxErrorGet (unsigned long ulBase)

Gets current receiver errors.

Parameters *ulBase* is the base address of the UART port.

This function returns the current state of each of the 4 receiver error sources. The returned errors are equivalent to the four error bits returned via the previous call to UARTCharGet() or UARTCharGetNonBlocking() with the exception that the overrun error is set immediately the overrun occurs rather than when a character is next read.

Returns Returns a logical OR combination of the receiver error flags, **UART_RXERROR_FRAMING**, **UART_RXERROR_PARITY**, **UART_RXERROR_BREAK** and **UART_RXERROR_OVERRUN**.

6.1.2.23 void UARTRXIntRegister (unsigned long ulBase, void(*)(void) pfnHandler)

Registers an interrupt handler for a UART RX interrupt.

Parameters *ulBase* is the base address of the UART port.

---

pfnHandler is a pointer to the function to be called when the UART interrupt occurs.

This function does the actual registering of the interrupt handler. This will enable the global interrupt in the interrupt controller; specific UART interrupts must be enabled via UARTIntEnable(). It is the interrupt handler's responsibility to clear the interrupt source.

See Also IntRegister() for important information about registering interrupt handlers. Returns None.

References IntEnable(), and IntRegister().

6.1.2.24 void UARTRXIntUnregister (unsigned long ulBase)

Unregisters an interrupt handler for a UART RX interrupt.

Parameters *ulBase* is the base address of the UART port.

This function does the actual unregistering of the interrupt handler. It will clear the handler to be called when a UART interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also IntRegister() for important information about registering interrupt handlers. Returns None.

References IntDisable(), and IntUnregister().

6.1.2.25 tBoolean UARTSpaceAvail (

unsigned long ulBase)

Determines if there is any space in the transmit FIFO.

Parameters *ulBase* is the base address of the UART port.

This function returns a flag indicating whether or not there is space available in the transmit FIFO.

Returns Returns **true** if there is space available in the transmit FIFO or **false** if there is no space available in the transmit FIFO.

6.1.2.26 unsigned long UARTTxIntModeGet (

unsigned long ulBase)

Returns the current operating mode for the UART transmit interrupt.

Parameters *ulBase* is the base address of the UART port.

This function returns the current operating mode for the UART transmit interrupt. The return value will be **UART_TXINT_MODE_EOT** if the transmit interrupt is currently set to be asserted once the transmitter is completely idle - the transmit FIFO is empty and all bits, including any stop bits, have cleared the transmitter. The return value will be **UART_TXINT_MODE_FIFO** if the interrupt is set to be asserted based upon the level of the transmit FIFO.

Returns Returns **UART_TXINT_MODE_FIFO** or **UART_TXINT_MODE_EOT**.

6.1.2.27 void UARTTxIntModeSet (

unsigned long ulBase,

unsigned long ulMode)

Sets the operating mode for the UART transmit interrupt.

Parameters *ulBase* is the base address of the UART port.

---

ulMode is the operating mode for the transmit interrupt. It may be **UART_TXINT_MODE_EOT** to trigger interrupts when the transmitter is idle or **UART_TXINT_MODE_FIFO** to trigger based on the current transmit FIFO level.

This function allows the mode of the UART transmit interrupt to be set. By default, the transmit interrupt is asserted when the FIFO level falls past a threshold set via a call to UARTFIFOLevelSet(). Alternatively, if this function is called with *ulMode* set to **UART_TXINT_MODE_EOT**, the transmit interrupt will only be asserted once the transmitter is completely idle - the transmit FIFO is empty and all bits, including any stop bits, have cleared the transmitter.

Returns None.

6.1.2.28 void UARTTXIntRegister (

unsigned long ulBase,

void(*)(void) pfnHandler)

Registers an interrupt handler for a UART TX interrupt.

Parameters *ulBase* is the base address of the UART port.

---

pfnHandler is a pointer to the function to be called when the UART interrupt occurs.

This function does the actual registering of the interrupt handler. This will enable the global interrupt in the interrupt controller; specific UART interrupts must be enabled via UARTIntEnable(). It is the interrupt handler's responsibility to clear the interrupt source.

See Also IntRegister() for important information about registering interrupt handlers. Returns None.

References IntEnable(), and IntRegister().

6.1.2.29 void UARTTXIntUnregister (

unsigned long ulBase)

Unregisters an interrupt handler for a UART TX interrupt.

Parameters *ulBase* is the base address of the UART port.

This function does the actual unregistering of the interrupt handler. It will clear the handler to be called when a UART interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also IntRegister() for important information about registering interrupt handlers. Returns None.

References IntDisable(), and IntUnregister().

The following example shows how to use the UART API to initialize the UART, transmit characters, and receive characters.

// // Initialize the UART. Set the baud rate, number of data bits, turn off // parity, number of stop bits, and stick mode. // UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 38400, (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

// // Enable the UART. // UARTEnable(UART0_BASE);

// // Check for characters. This will spin here until a character is placed // into the receive FIFO. // while(!UARTCharsAvail(UART0_BASE)) { }

// // Get the character(s) in the receive FIFO. // while(UARTCharGetNonBlocking(UART0_BASE)) { }

// // Put a character in the output buffer. // UARTCharPut(UART0_BASE, 'c'));

// // Disable the UART. // UARTDisable(UART0_BASE);

7 USB Controller

 The USB APIs provide a set of functions that are used to access the USB controller. The APIs are split into groups according to the functionality they provide. The groups are the following: USBDev, USBHost, USBEndpoint, and USBFIFO. The APIs in the USBDev group are only used with applications that implement a USB Device. The APIs in the USBHost group are only used in applications that implement a USB Host. The remainder of the APIs are used for both USB host and USB device controllers. The USBEndpoint APIs are used to configure and access the endpoints while the USBFIFO APIs are used to configure the size and location of the FIFOs.

7.1 API Functions

Functions

uint32_t USBDevAddrGet (uint32_t ulBase)

void USBDevAddrSet (uint32_t ulBase, uint32_t ulAddress)

void USBDevConnect (uint32_t ulBase)

void USBDevDisconnect (uint32_t ulBase)

void USBDevEndpointConfigGet (uint32_t ulBase, uint32_t ulEndpoint, uint32_t *pulMaxPacketSize, uint32_t *pulFlags)

void USBDevEndpointConfigSet (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulMaxPacketSize, uint32_t ulFlags)

void USBDevEndpointDataAck (uint32_t ulBase, uint32_t ulEndpoint, tBoolean bIsLastPacket)

void USBDevEndpointStall (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFlags)

void USBDevEndpointStallClear (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFlags)

void USBDevEndpointStatusClear (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFlags)

void USBDevMode (uint32_t ulBase)

uint32_t USBEndpointDataAvail (uint32_t ulBase, uint32_t ulEndpoint)

int32_t USBEndpointDataGet (uint32_t ulBase, uint32_t ulEndpoint, uint8_t *pucData, uint32_t *pulSize)

int32_t USBEndpointDataPut (uint32_t ulBase, uint32_t ulEndpoint, uint8_t *pucData, uint32_t ulSize)

int32_t USBEndpointDataSend (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulTransType)

void USBEndpointDataToggleClear (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFlags)

void USBEndpointDMAChannel (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulChannel)

void USBEndpointDMADisable (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFlags)

void USBEndpointDMAEnable (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFlags)

void USBEndpointPacketCountSet (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulCount)

uint32_t USBEndpointStatus (uint32_t ulBase, uint32_t ulEndpoint)

uint32_t USBFIFOAddrGet (uint32_t ulBase, uint32_t ulEndpoint)

void USBFIFOConfigGet (uint32_t ulBase, uint32_t ulEndpoint, uint32_t *pulFIFOAddress, uint32_t *pulFIFOSize, uint32_t ulFlags)

void USBFIFOConfigSet (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFIFOAddress, uint32_t ulFIFOSize, uint32_t ulFlags)

void USBFIFOFlush (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFlags)

uint32_t USBFrameNumberGet (uint32_t ulBase)

uint32_t USBHostAddrGet (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFlags)

void USBHostAddrSet (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulAddr, uint32_t ulFlags)

void USBHostEndpointConfig (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulMaxPayload, uint32_t ulNAKPollInterval, uint32_t ulTargetEndpoint, uint32_t ulFlags)

void USBHostEndpointDataAck (uint32_t ulBase, uint32_t ulEndpoint)

void USBHostEndpointDataToggle (uint32_t ulBase, uint32_t ulEndpoint, tBoolean bDataToggle, uint32_t ulFlags)

void USBHostEndpointStatusClear (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFlags)

uint32_t USBHostHubAddrGet (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulFlags)

void USBHostHubAddrSet (uint32_t ulBase, uint32_t ulEndpoint, uint32_t ulAddr, uint32_t ulFlags)

void USBHostMode (uint32_t ulBase)

void USBHostPwrConfig (uint32_t ulBase, uint32_t ulFlags)

void USBHostPwrDisable (uint32_t ulBase)

void USBHostPwrEnable (uint32_t ulBase)

void USBHostPwrFaultDisable (uint32_t ulBase)

void USBHostPwrFaultEnable (uint32_t ulBase)

void USBHostRequestIN (uint32_t ulBase, uint32_t ulEndpoint)

void USBHostRequestINClear (uint32_t ulBase, uint32_t ulEndpoint)

void USBHostRequestStatus (uint32_t ulBase)

void USBHostReset (uint32_t ulBase, tBoolean bStart)

void USBHostResume (uint32_t ulBase, tBoolean bStart)

uint32_t USBHostSpeedGet (uint32_t ulBase)

void USBHostSuspend (uint32_t ulBase)

void USBIntDisable (uint32_t ulBase, uint32_t ulFlags)

void USBIntDisableControl (uint32_t ulBase, uint32_t ulFlags)

void USBIntDisableEndpoint (uint32_t ulBase, uint32_t ulFlags)

void USBIntEnable (uint32_t ulBase, uint32_t ulFlags)

void USBIntEnableControl (uint32_t ulBase, uint32_t ulFlags)

void USBIntEnableEndpoint (uint32_t ulBase, uint32_t ulFlags)

void USBIntRegister (uint32_t ulBase, void (*pfnHandler)(void))

uint32_t USBIntStatus (uint32_t ulBase, uint32_t *pulIntStatusEP)

uint32_t USBIntStatusControl (uint32_t ulBase)

uint32_t USBIntStatusEndpoint (uint32_t ulBase)

void USBIntUnregister (uint32_t ulBase)

uint32_t USBModeGet (uint32_t ulBase)

uint32_t USBNumEndpointsGet (uint32_t ulBase)

void USBOTGMode (uint32_t ulBase)

void USBOTGSessionRequest (uint32_t ulBase, tBoolean bStart)

void USBPHYPowerOff (uint32_t ulBase)

void USBPHYPowerOn (uint32_t ulBase)

7.1.1 Detailed Description

The USB APIs provide all of the functions needed by an application to implement a USB device or USB host stack. The APIs abstract the IN/OUT nature of endpoints based on the type of USB controller that is in use. Any API that uses the IN/OUT terminology will comply with the standard USB interpretation of these terms. For example, an OUT endpoint on a microcontroller that has only a device interface will actually receive data on this endpoint, while a microcontroller that has a host interface will actually transmit data on an OUT endpoint.

Another important fact to understand is that all endpoints in the USB controller, whether host or device, have two "sides" to them. This allows each endpoint to both transmit and receive data. An application can use a single endpoint for both and IN and OUT transactions. For example: In device mode, endpoint 1 could be configured to have BULK IN and BULK OUT handled by endpoint 1. It

is important to note that the endpoint number used is the endpoint number reported to the host. In host mode, the application can use an endpoint to communicate with both IN and OUT endpoints of different types as well. For example: Endpoint 2 could be used to communicate with one device's interrupt IN endpoint and another device's bulk OUT endpoint at the same time. This effectively gives the application one dedicated control endpoint for IN or OUT control transactions on endpoint 0, and three IN endpoints and three OUT endpoints.

The USB controller has a configurable FIFO. The overall size of the FIFO RAM is 4096 bytes. It is important to note that the first 64 bytes of this memory are dedicated to endpoint 0 for control transactions. The remaining 4032 bytes are configurable however the application desires. The FIFO configuration is usually set at the beginning of the application and not modified once the USB controller is in use. The FIFO configuration uses the USBFIFOConfig() API to set the starting address and the size of the FIFOs that are dedicated to each endpoint.

Example: FIFO Configuration

0-64 - endpoint 0 IN/OUT (64 bytes).

64-576 - endpoint 1 IN (512 bytes).

576-1088 - endpoint 1 OUT (512 bytes).

1088-1600 - endpoint 2 IN (512 bytes). // // FIFO for endpoint 1 IN starts at address 64 and is 512 bytes in size. // USBFIFOConfig(USB0_BASE, USB_EP_1, 64, USB_FIFO_SZ_512, USB_EP_DEV_IN);

// // FIFO for endpoint 1 OUT starts at address 576 and is 512 bytes in size. // USBFIFOConfig(USB0_BASE, USB_EP_1, 576, USB_FIFO_SZ_512, USB_EP_DEV_OUT);

// // FIFO for endpoint 2 IN starts at address 1088 and is 512 bytes in size. // USBFIFOConfig(USB0_BASE, USB_EP_2, 1088, USB_FIFO_SZ_512, USB_EP_DEV_IN);

7.1.2 Function Documentation

7.1.2.1

uint32_t ulBase) Returns the current device address in device mode.

Parameters *ulBase* specifies the USB module base address.

This function returns the current device address. This address was set by a call to USBDevAddrSet().

Prototype:

```
USBDevAddrGet uint32_t USBDevAddrGet(
```

Note This function should only be called in device mode. Returns The current device address.

7.1.2.2 void USBDevAddrSet (

uint32_t ulBase,

uint32_t ulAddress)

Sets the address in device mode.

Parameters *ulBase* specifies the USB module base address.

---

ulAddress is the address to use for a device.

This function configures the device address on the USB bus. This address was likely received via a SET ADDRESS command from the host controller.

Note This function should only be called in device mode. Returns None.

7.1.2.3 void USBDevConnect (

uint32_t ulBase)

Connects the USB controller to the bus in device mode.

Parameters *ulBase* specifies the USB module base address.

This function causes the soft connect feature of the USB controller to be enabled. Call USBDevDisconnect() to remove the USB device from the bus.

Note This function should only be called in device mode. Returns None.

7.1.2.4 void USBDevDisconnect (

uint32_t ulBase)

Removes the USB controller from the bus in device mode.

Parameters *ulBase* specifies the USB module base address.

This function causes the soft connect feature of the USB controller to remove the device from the USB bus. A call to USBDevConnect() is needed to reconnect to the bus.

Note This function should only be called in device mode. Returns None.

7.1.2.5 void USBDevEndpointConfigGet (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t * pulMaxPacketSize,

uint32_t * pulFlags)

Gets the current configuration for an endpoint.

Parameters *ulBase* specifies the USB module base address.

ulEndpoint is the endpoint to access.

pulMaxPacketSize is a pointer which is written with the maximum packet size for this endpoint.

pulFlags is a pointer which is written with the current endpoint settings. On entry to the function, this pointer must contain either **USB_EP_DEV_IN** or **USB_EP_DEV_OUT** to indicate whether the IN or OUT endpoint is to be queried.

This function returns the basic configuration for an endpoint in device mode. The values returned in ∗*pulMaxPacketSize* and ∗*pulFlags* are equivalent to the *ulMaxPacketSize* and *ulFlags* previously passed to USBDevEndpointConfigSet() for this endpoint.

Note This function should only be called in device mode. Returns None.

7.1.2.6 void USBDevEndpointConfigSet (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulMaxPacketSize,

uint32_t ulFlags)

Sets the configuration for an endpoint.

Parameters *ulBase* specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulMaxPacketSize is the maximum packet size for this endpoint.

ulFlags are used to configure other endpoint settings.

This function sets the basic configuration for an endpoint in device mode. Endpoint zero does not have a dynamic configuration, so this function should not be called for endpoint zero. The *ulFlags* parameter determines some of the configuration while the other parameters provide the rest.

The **USB_EP_MODE_** flags define what the type is for the given endpoint.

USB_EP_MODE_CTRL is a control endpoint.

USB_EP_MODE_ISOC is an isochronous endpoint.

USB_EP_MODE_BULK is a bulk endpoint.

USB_EP_MODE_INT is an interrupt endpoint.

The **USB_EP_DMA_MODE_** flags determine the type of DMA access to the endpoint data FIFOs. The choice of the DMA mode depends on how the DMA controller is configured and how it is

being used. See the “Using USB with the uDMA Controller” section for more information on DMA configuration.

When configuring an IN endpoint, the **USB_EP_AUTO_SET** bit can be specified to cause the automatic transmission of data on the USB bus as soon as *ulMaxPacketSize* bytes of data are written into the FIFO for this endpoint. This option is commonly used with DMA as no interaction is required to start the transmission of data.

When configuring an OUT endpoint, the **USB_EP_AUTO_REQUEST** bit is specified to trigger the request for more data once the FIFO has been drained enough to receive *ulMaxPacketSize* more bytes of data. Also for OUT endpoints, the **USB_EP_AUTO_CLEAR** bit can be used to clear the data packet ready flag automatically once the data has been read from the FIFO. If this option is not used, this flag must be manually cleared via a call to USBDevEndpointStatusClear(). Both of these settings can be used to remove the need for extra calls when using the controller in DMA mode.

Note This function should only be called in device mode. Returns None.

7.1.2.7 void USBDevEndpointDataAck (

uint32_t ulBase,

uint32_t ulEndpoint,

tBoolean bIsLastPacket)

Acknowledge that data was read from the given endpoint’s FIFO in device mode.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

bIsLastPacket indicates if this packet is the last one.

This function acknowledges that the data was read from the endpoint’s FIFO. The *bIsLastPacket* parameter is set to a **true** value if this is the last in a series of data packets on endpoint zero. The *bIsLastPacket* parameter is not used for endpoints other than endpoint zero. This call can be used if processing is required between reading the data and acknowledging that the data has been read.

Note This function should only be called in device mode. Returns None.

7.1.2.8 void USBDevEndpointStall (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFlags)

Stalls the specified endpoint in device mode.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint specifies the endpoint to stall.

ulFlags specifies whether to stall the IN or OUT endpoint.

This function causes the endpoint number passed in to go into a stall condition. If the *ulFlags* parameter is **USB_EP_DEV_IN**, then the stall is issued on the IN portion of this endpoint. If the *ulFlags* parameter is **USB_EP_DEV_OUT**, then the stall is issued on the OUT portion of this endpoint.

Note This function should only be called in device mode. Returns None.

7.1.2.9 void USBDevEndpointStallClear (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFlags)

Clears the stall condition on the specified endpoint in device mode.

Parameters *ulBase* specifies the USB module base address.

ulEndpoint specifies which endpoint to remove the stall condition.

ulFlags specifies whether to remove the stall condition from the IN or the OUT portion of this endpoint.

This function causes the endpoint number passed in to exit the stall condition. If the *ulFlags* parameter is **USB_EP_DEV_IN**, then the stall is cleared on the IN portion of this endpoint. If the *ulFlags* parameter is **USB_EP_DEV_OUT**, then the stall is cleared on the OUT portion of this endpoint.

Note This function should only be called in device mode. Returns None.

7.1.2.10 void USBDevEndpointStatusClear (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFlags)

Clears the status bits in this endpoint in device mode.

Parameters *ulBase* specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFlags are the status bits that should be cleared.

This function clears the status of any bits that are passed in the *ulFlags* parameter. The *ulFlags* parameter can take the value returned from the USBEndpointStatus() call.

Note This function should only be called in device mode. Returns None.

7.1.2.11 void USBDevMode (

uint32_t ulBase)

Change the mode of the USB controller to device.

Parameters *ulBase* specifies the USB module base address.

This function changes the mode of the USB controller to device mode.

Note This function should only be called on microcontrollers that support OTG operation and have the DEVMODOTG bit in the USBGPCS register. Returns None.

7.1.2.12 uint32_t USBEndpointDataAvail (

uint32_t ulBase,

uint32_t ulEndpoint)

Determine the number of bytes of data available in a given endpoint's FIFO.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

This function returns the number of bytes of data currently available in the FIFO for the given receive (OUT) endpoint. It may be used prior to calling USBEndpointDataGet() to determine the size of buffer required to hold the newly-received packet.

Returns This call returns the number of bytes available in a given endpoint FIFO.

7.1.2.13 int32_t USBEndpointDataGet (

uint32_t ulBase,

uint32_t ulEndpoint,

uint8_t * pucData,

uint32_t * pulSize)

Retrieves data from the given endpoint's FIFO.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

pucData is a pointer to the data area used to return the data from the FIFO.

pulSize is initially the size of the buffer passed into this call via the *pucData* parameter. It is set to the amount of data returned in the buffer.

This function returns the data from the FIFO for the given endpoint. The *pulSize* parameter should indicate the size of the buffer passed in the *pulData* parameter. The data in the *pulSize* parameter is changed to match the amount of data returned in the *pucData* parameter. If a zero-byte packet is received, this call does not return an error but instead just returns a zero in the *pulSize* parameter. The only error case occurs when there is no data packet available.

Returns This call returns 0, or -1 if no packet was received.

7.1.2.14 int32_t USBEndpointDataPut (

uint32_t ulBase,

uint32_t ulEndpoint,

uint8_t * pucData,

uint32_t ulSize)

Puts data into the given endpoint's FIFO.

Parameters *ulBase* specifies the USB module base address.

ulEndpoint is the endpoint to access.

pucData is a pointer to the data area used as the source for the data to put into the FIFO.

ulSize is the amount of data to put into the FIFO.

This function puts the data from the *pucData* parameter into the FIFO for this endpoint. If a packet is already pending for transmission, then this call does not put any of the data into the FIFO and returns -1. Care should be taken to not write more data than can fit into the FIFO allocated by the call to USBFIFOConfigSet().

Returns This call returns 0 on success, or -1 to indicate that the FIFO is in use and cannot be written.

7.1.2.15 int32_t USBEndpointDataSend (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulTransType)

Starts the transfer of data from an endpoint's FIFO.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

ulTransType is set to indicate what type of data is being sent.

This function starts the transfer of data from the FIFO for a given endpoint. This function should be called if the **USB_EP_AUTO_SET** bit was not enabled for the endpoint. Setting the *ulTransType* parameter allows the appropriate signaling on the USB bus for the type of transaction being requested. The *ulTransType* parameter should be one of the following:

USB_TRANS_OUT for OUT transaction on any endpoint in host mode.

USB_TRANS_IN for IN transaction on any endpoint in device mode.

USB_TRANS_IN_LAST for the last IN transaction on endpoint zero in a sequence of IN transactions.

USB_TRANS_SETUP for setup transactions on endpoint zero.

USB_TRANS_STATUS for status results on endpoint zero.

Returns This call returns 0 on success, or -1 if a transmission is already in progress.

7.1.2.16 void USBEndpointDataToggleClear (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFlags)

Sets the Data toggle on an endpoint to zero.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint specifies the endpoint to reset the data toggle.

---

ulFlags specifies whether to access the IN or OUT endpoint.

This function causes the USB controller to clear the data toggle for an endpoint. This call is not valid for endpoint zero and can be made with host or device controllers.

The *ulFlags* parameter should be one of **USB_EP_HOST_OUT**, **USB_EP_HOST_IN**, **USB_EP_DEV_OUT**, or **USB_EP_DEV_IN**.

Returns None.

7.1.2.17 void USBEndpointDMAChannel (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulChannel)

Sets the DMA channel to use for a given endpoint.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint specifies which endpoint's FIFO address to return.

---

ulChannel specifies which DMA channel to use for which endpoint.

This function is used to configure which DMA channel to use with a given endpoint. Receive DMA channels can only be used with receive endpoints and transmit DMA channels can only be used with transmit endpoints. As a result, the 3 receive and 3 transmit DMA channels can be mapped to any endpoint other than 0. The values that should be passed into the *ulChannel* value are the UDMA_CHANNEL_USBEP∗ values defined in udma.h.

Note This function only has an effect on microcontrollers that have the ability to change the DMA channel for an endpoint. Calling this function on other devices has no effect. Returns None.

7.1.2.18 void USBEndpointDMADisable (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFlags)

Disable DMA on a given endpoint.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

ulFlags specifies which direction to disable.

This function disables DMA on a given endpoint to allow non-DMA USB transactions to generate interrupts normally. The ulFlags should be **USB_EP_DEV_IN** or **USB_EP_DEV_OUT**; all other bits are ignored.

Returns None.

7.1.2.19 void USBEndpointDMAEnable (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFlags)

Enable DMA on a given endpoint.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

ulFlags specifies which direction and what mode to use when enabling DMA.

This function will enable DMA on a given endpoint and set the mode according to the values in the *ulFlags* parameter. The *ulFlags* parameter should have **USB_EP_DEV_IN** or **USB_EP_DEV_OUT** set.

Returns None.

7.1.2.20 void USBEndpointPacketCountSet (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulCount)

Sets the number of packets to request when transferring multiple bulk packets.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint index to target for this write.

---

ulCount is the number of packets to request.

This function sets the number of consecutive bulk packets to request when transferring multiple bulk packets with DMA.

Note This feature is not available on all Tiva devices. Please check the data sheet to determine if the USB controller has a DMA controller or if it must use the uDMA controller for DMA transfers. Returns None.

7.1.2.21 uint32_t USBEndpointStatus (

uint32_t ulBase,

uint32_t ulEndpoint)

Returns the current status of an endpoint.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

This function returns the status of a given endpoint. If any of these status bits must be cleared, then the USBDevEndpointStatusClear() or the USBHostEndpointStatusClear() functions should be called.

The following are the status flags for host mode:

USB_HOST_IN_PID_ERROR - PID error on the given endpoint.

USB_HOST_IN_NOT_COMP - The device failed to respond to an IN request.

USB_HOST_IN_STALL - A stall was received on an IN endpoint.

USB_HOST_IN_DATA_ERROR - There was a CRC or bit-stuff error on an IN endpoint in Isochronous mode.

USB_HOST_IN_NAK_TO - NAKs received on this IN endpoint for more than the specified timeout period.

USB_HOST_IN_ERROR - Failed to communicate with a device using this IN endpoint.

USB_HOST_IN_FIFO_FULL - This IN endpoint's FIFO is full.

USB_HOST_IN_PKTRDY - Data packet ready on this IN endpoint.

USB_HOST_OUT_NAK_TO - NAKs received on this OUT endpoint for more than the specified timeout period.

USB_HOST_OUT_NOT_COMP - The device failed to respond to an OUT request.

USB_HOST_OUT_STALL - A stall was received on this OUT endpoint.

USB_HOST_OUT_ERROR - Failed to communicate with a device using this OUT endpoint.

USB_HOST_OUT_FIFO_NE - This endpoint's OUT FIFO is not empty.

USB_HOST_OUT_PKTPEND - The data transfer on this OUT endpoint has not completed.

USB_HOST_EP0_NAK_TO - NAKs received on endpoint zero for more than the specified timeout period.

USB_HOST_EP0_ERROR - The device failed to respond to a request on endpoint zero.

USB_HOST_EP0_IN_STALL - A stall was received on endpoint zero for an IN transaction.

USB_HOST_EP0_IN_PKTRDY - Data packet ready on endpoint zero for an IN transaction.

The following are the status flags for device mode:

USB_DEV_OUT_SENT_STALL - A stall was sent on this OUT endpoint.

USB_DEV_OUT_DATA_ERROR - There was a CRC or bit-stuff error on an OUT endpoint.

USB_DEV_OUT_OVERRUN - An OUT packet was not loaded due to a full FIFO.

USB_DEV_OUT_FIFO_FULL - The OUT endpoint's FIFO is full.

USB_DEV_OUT_PKTRDY - There is a data packet ready in the OUT endpoint's FIFO.

USB_DEV_IN_NOT_COMP - A larger packet was split up, more data to come.

USB_DEV_IN_SENT_STALL - A stall was sent on this IN endpoint.

USB_DEV_IN_UNDERRUN - Data was requested on the IN endpoint and no data was ready.

USB_DEV_IN_FIFO_NE - The IN endpoint's FIFO is not empty.

USB_DEV_IN_PKTPEND - The data transfer on this IN endpoint has not completed.

USB_DEV_EP0_SETUP_END - A control transaction ended before Data End condition was sent.

USB_DEV_EP0_SENT_STALL - A stall was sent on endpoint zero.

USB_DEV_EP0_IN_PKTPEND - The data transfer on endpoint zero has not completed.

USB_DEV_EP0_OUT_PKTRDY - There is a data packet ready in endpoint zero's OUT FIFO.

Returns The current status flags for the endpoint depending on mode.

7.1.2.22 uint32_t USBFIFOAddrGet (

uint32_t ulBase,

uint32_t ulEndpoint)

Returns the absolute FIFO address for a given endpoint.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint specifies which endpoint's FIFO address to return.

This function returns the actual physical address of the FIFO. This address is needed when the USB is going to be used with the uDMA controller and the source or destination address must be set to the physical FIFO address for a given endpoint.

Returns None.

7.1.2.23 void USBFIFOConfigGet (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t * pulFIFOAddress,

uint32_t * pulFIFOSize,

uint32_t ulFlags)

Returns the FIFO configuration for an endpoint.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

pulFIFOAddress is the starting address for the FIFO.

---

pulFIFOSize is the size of the FIFO as specified by one of the USB_FIFO_SZ_ values.

ulFlags specifies what information to retrieve from the FIFO configuration.

This function returns the starting address and size of the FIFO for a given endpoint. Endpoint zero does not have a dynamically configurable FIFO, so this function should not be called for endpoint zero. The *ulFlags* parameter specifies whether the endpoint's OUT or IN FIFO should be read. If in host mode, the *ulFlags* parameter should be **USB_EP_HOST_OUT** or **USB_EP_HOST_IN**, and if in device mode, the *ulFlags* parameter should be either **USB_EP_DEV_OUT** or **USB_EP_DEV_IN**.

Returns None.

7.1.2.24 void USBFIFOConfigSet (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFIFOAddress,

uint32_t ulFIFOSize,

uint32_t ulFlags)

Sets the FIFO configuration for an endpoint.

Parameters *ulBase* specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFIFOAddress is the starting address for the FIFO.

ulFIFOSize is the size of the FIFO specified by one of the USB_FIFO_SZ_ values.

ulFlags specifies what information to set in the FIFO configuration.

This function configures the starting FIFO RAM address and size of the FIFO for a given endpoint. Endpoint zero does not have a dynamically configurable FIFO, so this function should not be called for endpoint zero. The *ulFIFOSize* parameter should be one of the values in the **USB_FIFO_SZ_** values. If the endpoint is going to use double buffering, it should use the values with the **_DB** at the end of the value. For example, use **USB_FIFO_SZ_16_DB** to configure an endpoint to have a 16- byte, double-buffered FIFO. If a double-buffered FIFO is used, then the actual size of the FIFO is twice the size indicated by the *ulFIFOSize* parameter. For example, the **USB_FIFO_SZ_16_DB** value uses 32 bytes of the USB controller's FIFO memory.

The *ulFIFOAddress* value should be a multiple of 8 bytes and directly indicates the starting address in the USB controller's FIFO RAM. For example, a value of 64 indicates that the FIFO should start 64 bytes into the USB controller's FIFO memory. The *ulFlags* value specifies whether the endpoint's OUT or IN FIFO should be configured. If in host mode, use **USB_EP_HOST_OUT** or **USB_EP_HOST_IN**, and if in device mode, use **USB_EP_DEV_OUT** or **USB_EP_DEV_IN**.

Returns None.

7.1.2.25 void USBFIFOFlush (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFlags)

Forces a flush of an endpoint's FIFO.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

ulFlags specifies if the IN or OUT endpoint should be accessed.

This function forces the USB controller to flush out the data in the FIFO. The function can be called with either host or device controllers and requires the *ulFlags* parameter be one of **USB_EP_HOST_OUT**, **USB_EP_HOST_IN**, **USB_EP_DEV_OUT**, or **USB_EP_DEV_IN**.

Returns None.

7.1.2.26 uint32_t USBFrameNumberGet (

uint32_t ulBase)

Get the current frame number.

Parameters *ulBase* specifies the USB module base address.

This function returns the last frame number received.

Returns The last frame number received.

7.1.2.27 uint32_t USBHostAddrGet (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFlags)

Gets the current functional device address for an endpoint.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

ulFlags determines if this is an IN or an OUT endpoint.

This function returns the current functional address that an endpoint is using to communicate with a device. The *ulFlags* parameter determines if the IN or OUT endpoint's device address is returned.

722.7

ulNAKPollInterval is the either the NAK timeout limit or the polling interval, depending on the type of endpoint.

ulTargetEndpoint is the endpoint that the host endpoint is targeting.

ulFlags are used to configure other endpoint settings.

This function sets the basic configuration for the transmit or receive portion of an endpoint in host mode. The *ulFlags* parameter determines some of the configuration while the other parameters provide the rest. The *ulFlags* parameter determines whether this is an IN endpoint (**USB_EP_HOST_IN** or **USB_EP_DEV_IN**) or an OUT endpoint (**USB_EP_HOST_OUT** or **USB_EP_DEV_OUT**), whether this is a Full speed endpoint (**USB_EP_SPEED_FULL**) or a Low speed endpoint (**USB_EP_SPEED_LOW**).

The **USB_EP_MODE_** flags control the type of the endpoint.

USB_EP_MODE_CTRL is a control endpoint.

USB_EP_MODE_ISOC is an isochronous endpoint.

USB_EP_MODE_BULK is a bulk endpoint.

USB_EP_MODE_INT is an interrupt endpoint.

The *ulNAKPollInterval* parameter has different meanings based on the **USB_EP_MODE** value and whether or not this call is being made for endpoint zero or another endpoint. For endpoint zero or any Bulk endpoints, this value always indicates the number of frames to allow a device to NAK before considering it a timeout. If this endpoint is an isochronous or interrupt endpoint, this value is the polling interval for this endpoint.

For interrupt endpoints, the polling interval is simply the number of frames between polling an interrupt endpoint. For isochronous endpoints this value represents a polling interval of 2 ^ (*ulNAKPollInterval* - 1) frames. When used as a NAK timeout, the *ulNAKPollInterval* value specifies 2 ^ (*ulNAKPollInterval* - 1) frames before issuing a time out. There are two special time out values that can be specified when setting the *ulNAKPollInterval* value. The first is **MAX_NAK_LIMIT**, which is the maximum value that can be passed in this variable. The other is **DISABLE_NAK_LIMIT**, which indicates that there should be no limit on the number of NAKs.

The **USB_EP_DMA_MODE_** flags enable the type of DMA used to access the endpoint's data FIFOs. The choice of the DMA mode depends on how the DMA controller is configured and how it is being used. See the "Using USB with the uDMA Controller" section for more information on DMA configuration.

When configuring the OUT portion of an endpoint, the **USB_EP_AUTO_SET** bit is specified to cause the transmission of data on the USB bus to start as soon as the number of bytes specified by *ulMaxPayload* has been written into the OUT FIFO for this endpoint.

When configuring the IN portion of an endpoint, the **USB_EP_AUTO_REQUEST** bit can be specified to trigger the request for more data once the FIFO has been drained enough to fit *ulMaxPayload* bytes. The **USB_EP_AUTO_CLEAR** bit can be used to clear the data packet ready flag automatically once the data has been read from the FIFO. If this option is not used, this flag must be manually cleared via a call to USBDevEndpointStatusClear() or USBHostEndpointStatusClear().

Note This function should only be called in host mode. Returns None.

7.1.2.30 void USBHostEndpointDataAck (

uint32_t ulBase,

uint32_t ulEndpoint)

Acknowledge that data was read from the given endpoint's FIFO in host mode.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

This function acknowledges that the data was read from the endpoint's FIFO. This call is used if processing is required between reading the data and acknowledging that the data has been read.

Note This function should only be called in host mode. Returns None.

7.1.2.31 void USBHostEndpointDataToggle (

uint32_t ulBase,

uint32_t ulEndpoint,

tBoolean bDataToggle,

uint32_t ulFlags)

Sets the value data toggle on an endpoint in host mode.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint specifies the endpoint to reset the data toggle.

---

bDataToggle specifies whether to set the state to DATA0 or DATA1.

---

ulFlags specifies whether to set the IN or OUT endpoint.

This function is used to force the state of the data toggle in host mode. If the value passed in the *bDataToggle* parameter is **false**, then the data toggle is set to the DATA0 state, and if it is **true** it is set to the DATA1 state. The *ulFlags* parameter can be **USB_EP_HOST_IN** or **USB_EP_HOST_OUT** to access the desired portion of this endpoint. The *ulFlags* parameter is ignored for endpoint zero.

Note This function should only be called in host mode. Returns None.

7.1.2.32 void USBHostEndpointStatusClear (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFlags)

Clears the status bits in this endpoint in host mode.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

ulFlags are the status bits that should be cleared.

This function clears the status of any bits that are passed in the *ulFlags* parameter. The *ulFlags* parameter can take the value returned from the USBEndpointStatus() call.

Note This function should only be called in host mode. Returns None.

7.1.2.33 uint32_t USBHostHubAddrGet (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulFlags)

Get the current device hub address for this endpoint.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

ulFlags determines if this is an IN or an OUT endpoint.

This function returns the current hub address that an endpoint is using to communicate with a device. The *ulFlags* parameter determines if the device address for the IN or OUT endpoint is returned.

Note This function should only be called in host mode. Returns This function returns the current hub address being used by an endpoint.

7.1.2.34 void USBHostHubAddrSet (

uint32_t ulBase,

uint32_t ulEndpoint,

uint32_t ulAddr,

uint32_t ulFlags)

Set the hub address for the device that is connected to an endpoint.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

---

ulAddr is the hub address and port for the device using this endpoint. The hub address must be defined in bits 8 through 15 with the port number in bits 0 through 6.

---

ulFlags determines if this is an IN or an OUT endpoint.

This function configures the hub address for a device that is using this endpoint for communication. The *ulFlags* parameter determines if the device address for the IN or the OUT endpoint is configured by this call and sets the speed of the downstream device. Valid values are one of **USB_EP_HOST_OUT** or **USB_EP_HOST_IN** optionally ORed with **USB_EP_SPEED_LOW**.

Note This function should only be called in host mode. Returns None.

7.1.2.35 void USBHostMode (

uint32_t ulBase)

Change the mode of the USB controller to host.

Parameters *ulBase* specifies the USB module base address.

This function changes the mode of the USB controller to host mode.

Note This function should only be called on microcontrollers that support OTG operation and have the DEVMODOTG bit in the USBGPCS register. Returns None.

7.1.2.36 void USBHostPwrConfig (

uint32_t ulBase,

uint32_t ulFlags)

Sets the configuration for USB power fault.

Parameters *ulBase* specifies the USB module base address.

---

ulFlags specifies the configuration of the power fault.

This function controls how the USB controller uses its external power control pins (USBnPFLT and USBnEPEN). The flags specify the power fault level sensitivity, the power fault action, and the power enable level and source.

One of the following can be selected as the power fault level sensitivity:

USB_HOST_PWRFLT_LOW - An external power fault is indicated by the pin being driven low.

USB_HOST_PWRFLT_HIGH - An external power fault is indicated by the pin being driven high.

One of the following can be selected as the power fault action:

USB_HOST_PWRFLT_EP_NONE - No automatic action when power fault detected.

USB_HOST_PWRFLT_EP_TRI - Automatically tri-state the USBnEPEN pin on a power fault.

USB_HOST_PWRFLT_EP_LOW - Automatically drive USBnEPEN pin low on a power fault.

USB_HOST_PWRFLT_EP_HIGH - Automatically drive USBnEPEN pin high on a power fault.

One of the following can be selected as the power enable level and source:

USB_HOST_PWREN_MAN_LOW - USBnEPEN is driven low by the USB controller when USBHostPwrEnable() is called.

USB_HOST_PWREN_MAN_HIGH - USBnEPEN is driven high by the USB controller when USBHostPwrEnable() is called.

USB_HOST_PWREN_AUTOLOW - USBnEPEN is driven low by the USB controller automatically if USBOTGSessionRequest() has enabled a session.

USB_HOST_PWREN_AUTOHIGH - USBnEPEN is driven high by the USB controller automatically if USBOTGSessionRequest() has enabled a session.

On devices that support the VBUS glitch filter, the **USB_HOST_PWREN_FILTER** can be added to ignore small, short drops in VBUS level caused by high power consumption. This feature is mainly used to avoid causing VBUS errors caused by devices with high in-rush current.

Note This function must only be called on microcontrollers that support host mode or OTG operation. Returns None.

7.1.2.37 void USBHostPwrDisable (uint32_t ulBase)

Disables the external power pin.

Parameters *ulBase* specifies the USB module base address.

This function disables the USBnEPEN signal, which disables an external power supply in host mode operation.

Note This function must only be called in host mode. Returns None.

7.1.2.38 void USBHostPwrEnable (uint32_t ulBase)

Enables the external power pin.

Parameters *ulBase* specifies the USB module base address.

This function enables the USBnEPEN signal, which enables an external power supply in host mode operation.

Note This function must only be called in host mode. Returns None.

7.1.2.39 void USBHostPwrFaultDisable (uint32_t ulBase)

Disables power fault detection.

Parameters *ulBase* specifies the USB module base address.

This function disables power fault detection in the USB controller.

Note This function must only be called in host mode. Returns None.

7.1.2.40 void USBHostPwrFaultEnable (uint32_t ulBase)

Enables power fault detection.

Parameters *ulBase* specifies the USB module base address.

This function enables power fault detection in the USB controller. If the USBnPFLT pin is not in use, this function must not be used.

Note This function must only be called in host mode. Returns None.

7.1.2.41 void USBHostRequestIN (uint32_t ulBase, uint32_t ulEndpoint)

Schedules a request for an IN transaction on an endpoint in host mode.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

This function schedules a request for an IN transaction. When the USB device being communicated with responds with the data, the data can be retrieved by calling USBEndpointDataGet() or via a DMA transfer.

Note This function should only be called in host mode and only for IN endpoints. Returns None.

7.1.2.42 void USBHostRequestINClear (uint32_t ulBase, uint32_t ulEndpoint)

Clears a scheduled IN transaction for an endpoint in host mode.

Parameters *ulBase* specifies the USB module base address.

---

ulEndpoint is the endpoint to access.

This function clears a previously scheduled IN transaction if it is still pending. This function should be used to safely disable any scheduled IN transactions if the endpoint specified by *ulEndpoint* is reconfigured for communications with other devices.

Note This function should only be called in host mode and only for IN endpoints. Returns None.

7.1.2.43 void USBHostRequestStatus (uint32_t ulBase)

Issues a request for a status IN transaction on endpoint zero.

Parameters *ulBase* specifies the USB module base address.

This function is used to cause a request for a status IN transaction from a device on endpoint zero. This function can only be used with endpoint zero as that is the only control endpoint that supports this ability. This function is used to complete the last phase of a control transaction to a device and an interrupt is signaled when the status packet has been received.

Returns None.

7.1.2.44 void USBHostReset (uint32_t ulBase, tBoolean bStart)

Handles the USB bus reset condition.

Parameters *ulBase* specifies the USB module base address.

---

bStart specifies whether to start or stop signaling reset on the USB bus.

When this function is called with the *bStart* parameter set to **true**, this function causes the start of a reset condition on the USB bus. The caller should then delay at least 20ms before calling this function again with the *bStart* parameter set to **false**.

Note This function should only be called in host mode. Returns None.

7.1.2.45 void USBHostResume (uint32_t ulBase, tBoolean bStart)

Handles the USB bus resume condition.

Parameters *ulBase* specifies the USB module base address.

---

bStart specifies if the USB controller is entering or leaving the resume signaling state.

When in device mode, this function brings the USB controller out of the suspend state. This call should first be made with the *bStart* parameter set to **true** to start resume signaling. The device application should then delay at least 10ms but not more than 15ms before calling this function with the *bStart* parameter set to **false**.

When in host mode, this function signals devices to leave the suspend state. This call should first be made with the *bStart* parameter set to **true** to start resume signaling. The host application should then delay at least 20ms before calling this function with the *bStart* parameter set to **false**. This action causes the controller to complete the resume signaling on the USB bus.

Returns None.

7.1.2.46 uint32_t USBHostSpeedGet (

uint32_t ulBase)

Returns the current speed of the USB device connected.

Parameters *ulBase* specifies the USB module base address.

This function returns the current speed of the USB bus.

Note This function should only be called in host mode. Returns Returns either **USB_LOW_SPEED**, **USB_FULL_SPEED**, or **USB_UNDEF_SPEED**.

7.1.2.47 void USBHostSuspend (

uint32_t ulBase)

Puts the USB bus in a suspended state.

Parameters *ulBase* specifies the USB module base address.

When used in host mode, this function puts the USB bus in the suspended state.

Note This function should only be called in host mode. Returns None.

7.1.2.48 void USBIntDisable (

uint32_t ulBase,

uint32_t ulFlags)

Disables the sources for USB interrupts.

Parameters *ulBase* specifies the USB module base address.

---

ulFlags specifies which interrupts to disable.

This function disables the USB controller from generating the interrupts indicated by the *ulFlags* parameter. There are three groups of interrupt sources, IN Endpoints, OUT Endpoints, and general status changes, specified by **USB_INT_HOST_IN**, **USB_INT_HOST_OUT**, **USB_INT_DEV_IN**, **USB_INT_DEV_OUT**, and **USB_INT_STATUS**. If **USB_INT_ALL** is specified, then all interrupts are disabled.

Note WARNING: This API cannot be used on endpoint numbers greater than endpoint 3 so USBIntDisableControl() or USBIntDisableEndpoint() should be used instead. Returns None.

7.1.2.49 void USBIntDisableControl (

uint32_t ulBase,

uint32_t ulFlags)

Disable control interrupts on a given USB controller.

Parameters *ulBase* specifies the USB module base address.

---

ulFlags specifies which control interrupts to disable.

This function disables the control interrupts for the USB controller specified by the *ulBase* parameter. The *ulFlags* parameter specifies which control interrupts to disable. The flags passed in the *ulFlags* parameters should be the definitions that start with **USB_INTCTRL_**$*$ and not any other **USB_INT** flags.

Returns None.

7.1.2.50 void USBIntDisableEndpoint (

uint32_t ulBase,

uint32_t ulFlags)

Disable endpoint interrupts on a given USB controller.

Parameters *ulBase* specifies the USB module base address.

---

ulFlags specifies which endpoint interrupts to disable.

This function disables endpoint interrupts for the USB controller specified by the *ulBase* parameter. The *ulFlags* parameter specifies which endpoint interrupts to disable. The flags passed in the *ulFlags* parameters should be the definitions that start with **USB_INTEP_**$*$ and not any other **USB_INT** flags.

Returns None.

7.1.2.51 void USBIntEnable (

uint32_t ulBase,

uint32_t ulFlags)

Enables the sources for USB interrupts.

Parameters *ulBase* specifies the USB module base address.

---

ulFlags specifies which interrupts to enable.

This function enables the USB controller's ability to generate the interrupts indicated by the *ulFlags* parameter. There are three groups of interrupt sources, IN Endpoints, OUT Endpoints, and general status changes, specified by **USB_INT_HOST_IN**, **USB_INT_HOST_OUT**, **USB_INT_DEV_IN**, **USB_INT_DEV_OUT**, and **USB_STATUS**. If **USB_INT_ALL** is specified then all interrupts are enabled.

Note A call must be made to enable the interrupt in the main interrupt controller to receive interrupts. The USBIntRegister() API performs this controller-level interrupt enable. However if static interrupt handlers are used, then then a call to IntEnable() must be made in order to allow any USB interrupts to occur.

WARNING: This API cannot be used on endpoint numbers greater than endpoint 3 so USBIntEnableControl() or USBIntEnableEndpoint() should be used instead. Returns None.

7.1.2.52 void USBIntEnableControl (

uint32_t ulBase,

uint32_t ulFlags)

Enable control interrupts on a given USB controller.

Parameters *ulBase* specifies the USB module base address.

---

ulFlags specifies which control interrupts to enable.

This function enables the control interrupts for the USB controller specified by the *ulBase* parameter. The *ulFlags* parameter specifies which control interrupts to enable. The flags passed in the *ulFlags* parameters should be the definitions that start with **USB_INTCTRL_**$*$ and not any other **USB_INT** flags.

Returns None.

7.1.2.53 void USBIntEnableEndpoint (

uint32_t ulBase,

uint32_t ulFlags)

Enable endpoint interrupts on a given USB controller.

Parameters *ulBase* specifies the USB module base address.

---

ulFlags specifies which endpoint interrupts to enable.

This function enables endpoint interrupts for the USB controller specified by the *ulBase* parameter. The *ulFlags* parameter specifies which endpoint interrupts to enable. The flags passed in the *ulFlags* parameters should be the definitions that start with **USB_INTEP_**$*$ and not any other **USB_INT** flags.

Returns None.

7.1.2.54 void USBIntRegister (

uint32_t ulBase,

void(*)(void) pfnHandler)

Registers an interrupt handler for the USB controller.

Parameters *ulBase* specifies the USB module base address.

---

pfnHandler is a pointer to the function to be called when a USB interrupt occurs.

This function registers the handler to be called when a USB interrupt occurs and enables the global USB interrupt in the interrupt controller. The specific desired USB interrupts must be enabled via a separate call to USBIntEnable(). It is the interrupt handler's responsibility to clear the interrupt sources via calls to USBIntStatusControl() and USBIntStatusEndpoint().

See Also IntRegister() for important information about registering interrupt handlers. Returns None.

References IntEnable(), and IntRegister().

7.1.2.55 uint32_t USBIntStatus (

uint32_t ulBase,

uint32_t * pulIntStatusEP)

Returns the status of the USB interrupts.

Parameters *ulBase* specifies the USB module base address.

---

pulIntStatusEP is a pointer to the variable which holds the endpoint interrupt status from RXIS And TXIS.

This function reads the source of the interrupt for the USB controller. There are three groups of interrupt sources, IN Endpoints, OUT Endpoints, and general status changes. This call returns the current status for all of these interrupts. The bit values returned should be compared against the **USB_HOST_IN**, **USB_HOST_OUT**, **USB_HOST_EP0**, **USB_DEV_IN**, **USB_DEV_OUT**, and **USB_DEV_EP0** values.

Note This call clears the source of all of the general status interrupts.

WARNING: This API cannot be used on endpoint numbers greater than endpoint 3 so USBIntStatusControl() or USBIntStatusEndpoint() should be used instead. Returns Returns the status of the sources for the USB controller's interrupt.

7.1.2.56 uint32_t USBIntStatusControl (uint32_t ulBase)

Returns the control interrupt status on a given USB controller.

Parameters *ulBase* specifies the USB module base address.

This function reads control interrupt status for a USB controller. This call returns the current status for control interrupts only, the endpoint interrupt status is retrieved by calling USBIntStatusEndpoint(). The bit values returned should be compared against the **USB_INTCTRL_*** values.

The following are the meanings of all **USB_INCTRL_** flags and the modes for which they are valid. These values apply to any calls to USBIntStatusControl(), USBIntEnableControl(), and USBIntDisableControl(). Some of these flags are only valid in the following modes as indicated in the parentheses: Host, Device, and OTG.

USB_INTCTRL_ALL - A full mask of all control interrupt sources.

USB_INTCTRL_VBUS_ERR - A VBUS error has occurred (Host Only).

USB_INTCTRL_SESSION - Session Start Detected on A-side of cable (OTG Only).

USB_INTCTRL_SESSION_END - Session End Detected (Device Only)

USB_INTCTRL_DISCONNECT - Device Disconnect Detected (Host Only)

USB_INTCTRL_CONNECT - Device Connect Detected (Host Only)

USB_INTCTRL_SOF - Start of Frame Detected.

USB_INTCTRL_BABBLE - USB controller detected a device signaling past the end of a frame. (Host Only)

USB_INTCTRL_RESET - Reset signaling detected by device. (Device Only)

USB_INTCTRL_RESUME - Resume signaling detected.

USB_INTCTRL_SUSPEND - Suspend signaling detected by device (Device Only)

USB_INTCTRL_MODE_DETECT - OTG cable mode detection has completed (OTG Only)

USB_INTCTRL_POWER_FAULT - Power Fault detected. (Host Only)

Note This call clears the source of all of the control status interrupts. Returns Returns the status of the control interrupts for a USB controller.

7.1.2.57 uint32_t USBIntStatusEndpoint (

uint32_t ulBase)

Returns the endpoint interrupt status on a given USB controller.

Parameters *ulBase* specifies the USB module base address.

This function reads endpoint interrupt status for a USB controller. This call returns the current status for endpoint interrupts only, the control interrupt status is retrieved by calling USBIntStatusControl(). The bit values returned should be compared against the **USB_INTEP_**$*$ values. These values are grouped into classes for **USB_INTEP_HOST_**$*$ and **USB_INTEP_DEV_**$*$ values to handle both host and device modes with all endpoints.

Note This call clears the source of all of the endpoint interrupts. Returns Returns the status of the endpoint interrupts for a USB controller.

7.1.2.58 void USBIntUnregister (

uint32_t ulBase)

Unregisters an interrupt handler for the USB controller.

Parameters *ulBase* specifies the USB module base address.

This function unregisters the interrupt handler. This function also disables the USB interrupt in the interrupt controller.

See Also IntRegister() for important information about registering or unregistering interrupt handlers. Returns None.

References IntDisable(), and IntUnregister().

7.1.2.59 uint32_t USBModeGet (

uint32_t ulBase)

Returns the current operating mode of the controller.

Parameters *ulBase* specifies the USB module base address.

This function returns the current operating mode on USB controllers with OTG or Dual mode functionality.

For OTG controllers:

The function returns one of the following values on OTG controllers: **USB_OTG_MODE_ASIDE_HOST**, **USB_OTG_MODE_ASIDE_DEV**, **USB_OTG_MODE_BSIDE_HOST**, **USB_OTG_MODE_BSIDE_DEV**, **USB_OTG_MODE_NONE**.

USB_OTG_MODE_ASIDE_HOST indicates that the controller is in host mode on the A-side of the cable.

USB_OTG_MODE_ASIDE_DEV indicates that the controller is in device mode on the A-side of the cable.

USB_OTG_MODE_BSIDE_HOST indicates that the controller is in host mode on the B-side of the cable.

USB_OTG_MODE_BSIDE_DEV indicates that the controller is in device mode on the B-side of the cable. If an OTG session request is started with no cable in place, this mode is the default.

USB_OTG_MODE_NONE indicates that the controller is not attempting to determine its role in the system.

For Dual Mode controllers:

The function returns one of the following values: **USB_DUAL_MODE_HOST**, **USB_DUAL_MODE_DEVICE**, or **USB_DUAL_MODE_NONE**.

USB_DUAL_MODE_HOST indicates that the controller is acting as a host.

USB_DUAL_MODE_DEVICE indicates that the controller acting as a device.

USB_DUAL_MODE_NONE indicates that the controller is not active as either a host or device.

Returns Returns **USB_OTG_MODE_ASIDE_HOST**, **USB_OTG_MODE_ASIDE_DEV**, **USB_OTG_MODE_BSIDE_HOST**, **USB_OTG_MODE_BSIDE_DEV**, **USB_OTG_MODE_NONE**, **USB_DUAL_MODE_HOST**, **USB_DUAL_MODE_DEVICE**, or **USB_DUAL_MODE_NONE**.

7.1.2.60 uint32_t USBNumEndpointsGet (

uint32_t ulBase)

Returns the number of USB endpoint pairs on the device.

Parameters *ulBase* specifies the USB module base address.

This function returns the number of endpoint pairs supported by the USB controller corresponding to the passed base address. The value returned is the number of IN or OUT endpoints available and does not include endpoint 0 (the control endpoint). For example, if 15 is returned, there are 15 IN and 15 OUT endpoints available in addition to endpoint 0.

Returns Returns the number of IN or OUT endpoints available.

7.1.2.61 void USBOTGMode (

uint32_t ulBase)

Change the mode of the USB controller to OTG.

Parameters *ulBase* specifies the USB module base address.

This function changes the mode of the USB controller to OTG mode. This function is only valid on microcontrollers that have the OTG capabilities.

Returns None.

7.1.2.62 void USBOTGSessionRequest (

uint32_t ulBase,

tBoolean bStart)

Starts or ends a session.

Parameters *ulBase* specifies the USB module base address.

---

bStart specifies if this call starts or ends a session.

This function is used in OTG mode to start a session request or end a session. If the *bStart* parameter is set to **true**, then this function starts a session and if it is **false** it ends a session.

Returns None.

7.1.2.63 void USBPHYPowerOff (

uint32_t ulBase)

Powers off the USB PHY.

Parameters *ulBase* specifies the USB module base address.

This function powers off the USB PHY, reducing the current consuption of the device. While in the powered-off state, the USB controller is unable to operate.

Returns None.

7.1.2.64 void USBPHYPowerOn (

uint32_t ulBase)

Powers on the USB PHY.

Parameters *ulBase* specifies the USB module base address.

This function powers on the USB PHY, enabling it return to normal operation. By default, the PHY is powered on, so this function must only be called if USBPHYPowerOff() has previously been called.

Returns None. This example code makes the calls necessary to configure end point 1, in device mode, as a bulk IN end point. The first call configures end point 1 to have a maximum packet size of 64 bytes and makes it a bulk IN end point. The call to USBFIFOConfig() sets the starting address to 64 bytes in and 64 bytes long. It specifies **USB_EP_DEV_IN** to indicate that this is a device mode IN endpoint. The next two calls demonstrate how to fill the data FIFO for this endpoint and then have it scheduled for transmission on the USB bus. The USBEndpointDataPut() call puts data into the FIFO but does not actually start the data transmission. The USBEndpointDataSend() call will schedule the transmission to go out the next time the host controller requests data on this endpoint.

// // Configure Endpoint 1. // USBDevEndpointConfigSet(USB0_BASE, USB_EP_1, 64, DISABLE_NAK_LIMIT, USB_EP_MODE_BULK | USB_EP_DEV_IN);

// // Configure FIFO as a device IN endpoint FIFO starting at address 64 // and is 64 bytes in size. // USBFIFOConfig(USB0_BASE, USB_EP_1, 64, USB_FIFO_SZ_64, USB_EP_DEV_IN);

...

// // Put the data in the FIFO. // USBEndpointDataPut(USB0_BASE, USB_EP_1, pucData, 64);

// // Start the transmission of data. // USBEndpointDataSend(USB0_BASE, USB_EP_1, USB_TRANS_IN);

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265