F2837xS Peripheral Driver Library

USER'S GUIDE

F2837xS-DRL-UG-180

Copyright

Texas Instruments
12203 Southwest Freeway
Houston, TX 77477
http://www.ti.com/c2000

Revision Information

This is version 180 of this document, last updated on Fri Nov 6 16:28:24 CST 2015.

Table of Contents

1 Introduction

The Texas Instruments® ControlSUITE® Peripheral Driver Library is a set of drivers for accessing peripherals found on the Delfino 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 located within the "F2837xS_common" directory.

`driverlib/`	This directory contains the source code for the drivers.
`inc/`	This directory holds the part specific header files used for the direct register access programming model.
`inc/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.

2 Programming Model

2.1 Introduction

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.

2.2 Direct Register Access Model

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 several header files contained in the `inc` directory. By including the header files `inc/hw_types.h` and `inc/hw_memmap.h`, macros are available for accessing all registers on the Delfino devices. Individual bitfield access can easily be added by simply including the `inc/hw_peripheral.h` header file for the desired peripheral.

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:

```
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_CR0_R = 0x000005c7;
```

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

```
ulValue = (SSI0_CR0_R & SSI_CR0_SCR_M) >> SSI0_CR0_SCR_S;
```

The GPIO modules have many registers that do not have bit field definitions. For these registers, the register bits represent the individual GPIO pins; so bit zero in these registers corresponds to the **Px0** pin on the part (where **x** is replaced by a GPIO module letter), bit one corresponds to the **Px1** pin, and so on.

2.3 Software Driver Model

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_BASE, 50000000, SSI_FRF_MOTO_MODE_3,
                   SSI_MODE_MASTER, 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.

2.4 Combining The Models

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 Controller Area Network (CAN)

3.1 Introduction

The Controller Area Network (CAN) APIs provide a set of functions for accessing the Delfino CAN modules. Functions are provided to configure the CAN controllers, configure message objects, and manage CAN interrupts.

The Delfino CAN module provides hardware processing of the CAN data link layer. It can be configured with message filters and preloaded message data so that it can autonomously send and receive messages on the bus, and notify the application accordingly. It automatically handles generation and checking of CRCs, error processing, and retransmission of CAN messages.

The message objects are stored in the CAN controller and provide the main interface for the CAN module on the CAN bus. There are 32 message objects that can each be programmed to handle a separate message ID, or can be chained together for a sequence of frames with the same ID. The message identifier filters provide masking that can be programmed to match any or all of the message ID bits, and frame types.

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

3.2 API Functions

Data Structures

- tCANBitClkParms
- tCANMsgObject

Defines

- CAN_CLK_AUXCLKIN
- CAN_CLK_CPU_SYSCLKOUT
- CAN_CLK_EXT_OSC
- CAN_INT_ERROR
- CAN_INT_IE0
- CAN_INT_IE1
- CAN_INT_STATUS
- CAN_STATUS_BUS_OFF
- CAN_STATUS_EPASS
- CAN_STATUS_EWARN

- CAN_STATUS_LEC_ACK
- CAN_STATUS_LEC_BIT0
- CAN_STATUS_LEC_BIT1
- CAN_STATUS_LEC_CRC
- CAN_STATUS_LEC_FORM
- CAN_STATUS_LEC_MSK
- CAN_STATUS_LEC_NONE
- CAN_STATUS_LEC_STUFF
- CAN_STATUS_PDA
- CAN_STATUS_PERR
- CAN_STATUS_RXOK
- CAN_STATUS_TXOK
- CAN_STATUS_WAKE_UP
- MSG_OBJ_DATA_LOST
- MSG_OBJ_EXTENDED_ID
- MSG_OBJ_FIFO
- MSG_OBJ_NEW_DATA
- MSG_OBJ_NO_FLAGS
- MSG_OBJ_REMOTE_FRAME
- MSG_OBJ_RX_INT_ENABLE
- MSG_OBJ_STATUS_MASK
- MSG_OBJ_TX_INT_ENABLE
- MSG_OBJ_USE_DIR_FILTER
- MSG_OBJ_USE_EXT_FILTER
- MSG_OBJ_USE_ID_FILTER

Enumerations

- tCANIntStsReg
- tCANStsReg
- tMsgObjType

Functions

- uint32_t CANBitRateSet (uint32_t ui32Base, uint32_t ui32SourceClock, uint32_t ui32BitRate)
- void CANBitTimingGet (uint32_t ui32Base, tCANBitClkParms *pClkParms)
- void CANBitTimingSet (uint32_t ui32Base, tCANBitClkParms *pClkParms)
- void CANClkSourceSelect (uint32_t ui32Base, uint16_t ui16Source)
- void CANDisable (uint32_t ui32Base)
- void CANEnable (uint32_t ui32Base)
- bool CANErrCntrGet (uint32_t ui32Base, uint32_t *pui32RxCount, uint32_t *pui32TxCount)
- void CANGlobalIntClear (uint32_t ui32Base, uint32_t ui32IntFlags)
- void CANGlobalIntDisable (uint32_t ui32Base, uint32_t ui32IntFlags)
- void CANGlobalIntEnable (uint32_t ui32Base, uint32_t ui32IntFlags)

- bool CANGlobalIntstatusGet (uint32_t ui32Base, uint32_t ui32IntFlags)
- void CANInit (uint32_t ui32Base)
- void CANIntClear (uint32_t ui32Base, uint32_t ui32IntClr)
- void CANIntDisable (uint32_t ui32Base, uint32_t ui32IntFlags)
- void CANIntEnable (uint32_t ui32Base, uint32_t ui32IntFlags)
- void CANIntRegister (uint32_t ui32Base, unsigned char ucIntNumber, void (*pfnHandler)(void))
- uint32_t CANIntStatus (uint32_t ui32Base, tCANIntStsReg eIntStsReg)
- void CANIntUnregister (uint32_t ui32Base, unsigned char ucIntNumber)
- void CANMessageClear (uint32_t ui32Base, uint32_t ui32ObjID)
- void CANMessageGet (uint32_t ui32Base, uint32_t ui32ObjID, tCANMsgObject *pMsgObject, bool bClrPendingInt)
- void CANMessageSet (uint32_t ui32Base, uint32_t ui32ObjID, tCANMsgObject *pMsgObject, tMsgObjType eMsgType)
- bool CANRetryGet (uint32_t ui32Base)
- void CANRetrySet (uint32_t ui32Base, bool bAutoRetry)
- uint32_t CANStatusGet (uint32_t ui32Base, tCANStsReg eStatusReg)

3.2.1 Detailed Description

The CAN APIs provide all of the functions needed by the application to implement an interrupt-driven CAN stack. These functions may be used to control any of the available CAN ports on a Delfino microcontroller, and can be used with one port without causing conflicts with the other port.

The CAN module is disabled by default, so the the CANInit() function must be called before any other CAN functions are called. This call initializes the message objects to a safe state prior to enabling the controller on the CAN bus. Also, the bit timing values must be programmed prior to enabling the CAN controller. The CANBitTimingSet() function should be called with the appropriate bit timing values for the CAN bus. Once these two functions have been called, a CAN controller can be enabled using the CANEnable(), and later disabled using CANDisable() if needed. Calling CANDisable() does not reinitialize a CAN controller, so it can be used to temporarily remove a CAN controller from the bus.

The CAN controller is highly configurable and contains 32 message objects that can be programmed to automatically transmit and receive CAN messages under certain conditions. Message objects allow the application to perform some actions automatically without interaction from the microcontroller. Some examples of these actions are the following:

- Send a data frame immediately
- Send a data frame when a matching remote frame is seen on the CAN bus
- Receive a specific data frame
- Receive data frames that match a certain identifier pattern

To configure message objects to perform any of these actions, the application must first set up one of the 32 message objects using CANMessageSet(). This function must be used to configure a message object to send data, or to configure a message object to receive data. Each message object can be configured to generate interrupts on transmission or reception of CAN messages.

When data is received from the CAN bus, the application can use the CANMessageGet() function to read the received message. This function can also be used to read a message object that is already

configured in order to populate a message structure prior to making changes to the configuration of a message object. Reading the message object using this function will also clear any pending interrupt on the message object.

Once a message object has been configured using CANMessageSet(), it has allocated the message object and will continue to perform its programmed function unless it is released with a call to CANMessageClear(). The application is not required to clear out a message object before setting it with a new configuration, because each time CANMessageSet() is called, it will overwrite any previously programmed configuration.

The 32 message objects are identical except for priority. The lowest numbered message objects have the highest priority. Priority affects operation in two ways. First, if multiple actions are ready at the same time, the one with the highest priority message object will occur first. And second, when multiple message objects have interrupts pending, the highest priority will be presented first when reading the interrupt status. It is up to the application to manage the 32 message objects as a resource, and determine the best method for allocating and releasing them.

The CAN controller can generate interrupts on several conditions:

- When any message object transmits a message
- When any message object receives a message
- On warning conditions such as an error counter reaching a limit or occurrence of various bus errors
- On controller error conditions such as entering the bus-off state

An interrupt handler must be installed in order to process CAN interrupts. If dynamic interrupt configuration is desired, the CANIntRegister() can be used to register the interrupt handler. This will place the vector in a RAM-based vector table. However, if the application uses a pre-loaded vector table in flash, then the CAN controller handler should be entered in the appropriate slot in the vector table. In this case, CANIntRegister() is not needed, but the interrupt will need to be enabled on the host processor master interrupt controller using the IntEnable() function. The CAN module interrupts are enabled using the CANIntEnable() function. They can be disabled by using the CANIntDisable() function.

Once CAN interrupts are enabled, the handler will be invoked whenever a CAN interrupt is triggered. The handler can determine which condition caused the interrupt by using the CANIntStatus() function. Multiple conditions can be pending when an interrupt occurs, so the handler must be designed to process all pending interrupt conditions before exiting. Each interrupt condition must be cleared before exiting the handler. There are two ways to do this. The CANIntClear() function will clear a specific interrupt condition without further action required by the handler. However, the handler can also clear the condition by performing certain actions. If the interrupt is a status interrupt, the interrupt can be cleared by reading the status register with CANStatusGet(). If the interrupt is caused by one of the message objects, then it can be cleared by reading the message object using CANMessageGet().

There are several status registers that can be used to help the application manage the controller. The status registers are read using the CANStatusGet() function. There is a controller status register that provides general status information such as error or warning conditions. There are also several status registers that provide information about all of the message objects at once using a 32-bit bit map of the status, with one bit representing each message object. These status registers can be used to determine:

- Which message objects have unprocessed received data
- Which message objects have pending transmission requests

- Which message objects are allocated for use

3.2.2 Data Structure Documentation

3.2.2.1 tCANBitClkParms

Definition:

```
typedef struct
{
    uint16_t uSyncPropPhase1Seg;
    uint16_t uPhase2Seg;
    uint16_t uSJW;
    uint16_t uQuantumPrescaler;
}
tCANBitClkParms
```

Members:

uSyncPropPhase1Seg This value holds the sum of the Synchronization, Propagation, and Phase Buffer 1 segments, measured in time quanta. The valid values for this setting range from 2 to 16.

uPhase2Seg This value holds the Phase Buffer 2 segment in time quanta. The valid values for this setting range from 1 to 8.

uSJW This value holds the Resynchronization Jump Width in time quanta. The valid values for this setting range from 1 to 4.

uQuantumPrescaler This value holds the CAN_CLK divider used to determine time quanta. The valid values for this setting range from 1 to 1023.

Description:

This structure is used for encapsulating the values associated with setting up the bit timing for a CAN controller. The structure is used when calling the CANGetBitTiming and CANSetBitTiming functions.

3.2.2.2 tCANMsgObject

Definition:

```
typedef struct
{
    uint32_t ui32MsgID;
    uint32_t ui32MsgIDMask;
    uint32_t ui32Flags;
    uint32_t ui32MsgLen;
    unsigned char *pucMsgData;
}
tCANMsgObject
```

Members:

ui32MsgID The CAN message identifier used for 11 or 29 bit identifiers.

ui32MsgIDMask The message identifier mask used when identifier filtering is enabled.

ui32Flags This value holds various status flags and settings specified by tCANObjFlags.

ui32MsgLen This value is the number of bytes of data in the message object.

pucMsgData This is a pointer to the message object's data.

Description:
The structure used for encapsulating all the items associated with a CAN message object in the CAN controller.

3.2.3 Define Documentation

3.2.3.1 CAN_CLK_AUXCLKIN

Definition:
```
#define CAN_CLK_AUXCLKIN
```

Description:
This flag is used to clock the CAN controller with the clock from AUXCLKIN (from GPIO)

3.2.3.2 CAN_CLK_CPU_SYSCLKOUT

Definition:
```
#define CAN_CLK_CPU_SYSCLKOUT
```

Description:
This flag is used to clock the CAN controller Selected CPU SYSCLKOUT (CPU1.Sysclk or CPU2.Sysclk).

3.2.3.3 CAN_CLK_EXT_OSC

Definition:
```
#define CAN_CLK_EXT_OSC
```

Description:
This flag is used to clock the CAN controller with the X1/X2 oscillator clock.

3.2.3.4 CAN_INT_ERROR

Definition:
```
#define CAN_INT_ERROR
```

Description:
This flag is used to allow a CAN controller to generate error interrupts.

3.2.3.5 CAN_INT_IE0

Definition:
```
#define CAN_INT_IE0
```

Description:
This flag is used to allow a CAN controller to generate interrupts on interrupt line 0

3.2.3.6 CAN_INT_IE1

Definition:
```
#define CAN_INT_IE1
```

Description:
This flag is used to allow a CAN controller to generate interrupts on interrupt line 1

3.2.3.7 CAN_INT_STATUS

Definition:
```
#define CAN_INT_STATUS
```

Description:
This flag is used to allow a CAN controller to generate status interrupts.

3.2.3.8 CAN_STATUS_BUS_OFF

Definition:
```
#define CAN_STATUS_BUS_OFF
```

Description:
CAN controller has entered a Bus Off state.

3.2.3.9 CAN_STATUS_EPASS

Definition:
```
#define CAN_STATUS_EPASS
```

Description:
CAN controller error level has reached error passive level.

3.2.3.10 CAN_STATUS_EWARN

Definition:
```
#define CAN_STATUS_EWARN
```

Description:
CAN controller error level has reached warning level.

3.2.3.11 CAN_STATUS_LEC_ACK

Definition:

```
#define CAN_STATUS_LEC_ACK
```

Description:

An acknowledge error has occurred.

3.2.3.12 CAN_STATUS_LEC_BIT0

Definition:

```
#define CAN_STATUS_LEC_BIT0
```

Description:

The bus remained a bit level of 0 for longer than is allowed.

3.2.3.13 CAN_STATUS_LEC_BIT1

Definition:

```
#define CAN_STATUS_LEC_BIT1
```

Description:

The bus remained a bit level of 1 for longer than is allowed.

3.2.3.14 CAN_STATUS_LEC_CRC

Definition:

```
#define CAN_STATUS_LEC_CRC
```

Description:

A CRC error has occurred.

3.2.3.15 CAN_STATUS_LEC_FORM

Definition:

```
#define CAN_STATUS_LEC_FORM
```

Description:

A formatting error has occurred.

3.2.3.16 CAN_STATUS_LEC_MSK

Definition:

```
#define CAN_STATUS_LEC_MSK
```

Description:

This is the mask for the last error code field.

3.2.3.17 CAN_STATUS_LEC_NONE

Definition:

```
#define CAN_STATUS_LEC_NONE
```

Description:

There was no error.

3.2.3.18 CAN_STATUS_LEC_STUFF

Definition:

```
#define CAN_STATUS_LEC_STUFF
```

Description:

A bit stuffing error has occurred.

3.2.3.19 CAN_STATUS_PDA

Definition:

```
#define CAN_STATUS_PDA
```

Description:

CAN controller is in local power down mode.

3.2.3.20 CAN_STATUS_PERR

Definition:

```
#define CAN_STATUS_PERR
```

Description:

CAN controller has detected a parity error.

3.2.3.21 CAN_STATUS_RXOK

Definition:

```
#define CAN_STATUS_RXOK
```

Description:

A message was received successfully since the last read of this status.

3.2.3.22 CAN_STATUS_TXOK

Definition:

```
#define CAN_STATUS_TXOK
```

Description:

A message was transmitted successfully since the last read of this status.

3.2.3.23 CAN_STATUS_WAKE_UP

Definition:

```
#define CAN_STATUS_WAKE_UP
```

Description:

CAN controller has initiated a system wakeup.

3.2.3.24 MSG_OBJ_DATA_LOST

Definition:

```
#define MSG_OBJ_DATA_LOST
```

Description:

This indicates that data was lost since this message object was last read.

3.2.3.25 MSG_OBJ_EXTENDED_ID

Definition:

```
#define MSG_OBJ_EXTENDED_ID
```

Description:

This indicates that a message object will use or is using an extended identifier.

3.2.3.26 MSG_OBJ_FIFO

Definition:

```
#define MSG_OBJ_FIFO
```

Description:

This indicates that this message object is part of a FIFO structure and not the final message object in a FIFO.

3.2.3.27 MSG_OBJ_NEW_DATA

Definition:

```
#define MSG_OBJ_NEW_DATA
```

Description:

This indicates that new data was available in the message object.

3.2.3.28 MSG_OBJ_NO_FLAGS

Definition:

```
#define MSG_OBJ_NO_FLAGS
```

Description:

This indicates that a message object has no flags set.

3.2.3.29 MSG_OBJ_REMOTE_FRAME

Definition:

```
#define MSG_OBJ_REMOTE_FRAME
```

Description:

This indicates that a message object is a remote frame.

3.2.3.30 MSG_OBJ_RX_INT_ENABLE

Definition:

```
#define MSG_OBJ_RX_INT_ENABLE
```

Description:

This indicates that receive interrupts should be enabled, or are enabled.

3.2.3.31 MSG_OBJ_STATUS_MASK

Definition:

```
#define MSG_OBJ_STATUS_MASK
```

Description:

This define is used with the flag values to allow checking only status flags and not configuration flags.

3.2.3.32 MSG_OBJ_TX_INT_ENABLE

Definition:

```
#define MSG_OBJ_TX_INT_ENABLE
```

Description:

This definition is used with the tCANMsgObject ui32Flags value and indicates that transmit interrupts should be enabled, or are enabled.

3.2.3.33 MSG_OBJ_USE_DIR_FILTER

Definition:

```
#define MSG_OBJ_USE_DIR_FILTER
```

Description:

This indicates that a message object will use or is using filtering based on the direction of the transfer. If the direction filtering is used, then ID filtering must also be enabled.

3.2.3.34 MSG_OBJ_USE_EXT_FILTER

Definition:

```
#define MSG_OBJ_USE_EXT_FILTER
```

Description:

This indicates that a message object will use or is using message identifier filtering based on the extended identifier. If the extended identifier filtering is used, then ID filtering must also be enabled.

3.2.3.35 MSG_OBJ_USE_ID_FILTER

Definition:

```
#define MSG_OBJ_USE_ID_FILTER
```

Description:

This indicates that a message object will use or is using filtering based on the object's message identifier.

3.2.4 Enumeration Documentation

3.2.4.1 tCANIntStsReg

Description:

This data type is used to identify the interrupt status register. This is used when calling the CANIntStatus() function.

Enumerators:

CAN_INT_STS_CAUSE Read the CAN interrupt status information.

CAN_INT_STS_OBJECT Read a message object's interrupt status.

3.2.4.2 tCANStsReg

Description:

This data type is used to identify which of several status registers to read when calling the CANStatusGet() function.

Enumerators:

CAN_STS_CONTROL Read the full CAN controller status.

CAN_STS_TXREQUEST Read the full 32-bit mask of message objects with a transmit request set.

CAN_STS_NEWDAT Read the full 32-bit mask of message objects with new data available.

CAN_STS_MSGVAL Read the full 32-bit mask of message objects that are enabled.

3.2.4.3 tMsgObjType

Description:

This definition is used to determine the type of message object that will be set up via a call to the CANMessageSet() API.

Enumerators:

MSG_OBJ_TYPE_TX Transmit message object.

MSG_OBJ_TYPE_TX_REMOTE Transmit remote request message object.

MSG_OBJ_TYPE_RX Receive message object.

MSG_OBJ_TYPE_RX_REMOTE Receive remote request message object.

MSG_OBJ_TYPE_RXTX_REMOTE Remote frame receive remote, with auto-transmit message object.

3.2.5 Function Documentation

3.2.5.1 CANBitRateSet

This function is used to set the CAN bit timing values to a nominal setting based on a desired bit rate.

Prototype:

```
uint32_t
CANBitRateSet(uint32_t ui32Base,
              uint32_t ui32SourceClock,
              uint32_t ui32BitRate)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32SourceClock is the clock frequency for the CAN peripheral in Hz.

ui32BitRate is the desired bit rate.

Description:

This function will set the CAN bit timing for the bit rate passed in the *ui32BitRate* parameter based on the *ui32SourceClock* parameter. The CAN bit clock is calculated to be an average timing value that should work for most systems. If tighter timing requirements are needed, then the CANBitTimingSet() function is available for full customization of all of the CAN bit timing values. Since not all bit rates can be matched exactly, the bit rate is set to the value closest to the desired bit rate without being higher than the *ui32BitRate* value.

Returns:

This function returns the bit rate that the CAN controller was configured to use or it returns 0 to indicate that the bit rate was not changed because the requested bit rate was not valid.

3.2.5.2 CANBitTimingGet

Reads the current settings for the CAN controller bit timing.

Prototype:

```
void
CANBitTimingGet(uint32_t ui32Base,
                tCANBitClkParms *pClkParms)
```

Parameters:

ui32Base is the base address of the CAN controller.

pClkParms is a pointer to a structure to hold the timing parameters.

Description:

This function reads the current configuration of the CAN controller bit clock timing, and stores the resulting information in the structure supplied by the caller. Refer to CANBitTimingSet() for the meaning of the values that are returned in the structure pointed to by *pClkParms*.

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

Returns:

None.

3.2.5.3 CANBitTimingSet

Configures the CAN controller bit timing.

Prototype:

```
void
CANBitTimingSet(uint32_t ui32Base,
                tCANBitClkParms *pClkParms)
```

Parameters:

ui32Base is the base address of the CAN controller.

pClkParms points to the structure with the clock parameters.

Description:

Configures the various timing parameters for the CAN bus bit timing: Propagation segment, Phase Buffer 1 segment, Phase Buffer 2 segment, and the Synchronization Jump Width. The values for Propagation and Phase Buffer 1 segments are derived from the combination *pClkParms->uSyncPropPhase1Seg* parameter. Phase Buffer 2 is determined from the *pClkParms->uPhase2Seg* parameter. These two parameters, along with *pClkParms->uSJW* are based in units of bit time quanta. The actual quantum time is determined by the *pClkParms->uQuantumPrescaler* value, which specifies the divisor for the CAN module clock.

The total bit time, in quanta, will be the sum of the two Seg parameters, as follows:

bit_time_q = uSyncPropPhase1Seg + uPhase2Seg + 1

Note that the Sync_Seg is always one quantum in duration, and will be added to derive the correct duration of Prop_Seg and Phase1_Seg.

The equation to determine the actual bit rate is as follows:

CAN Clock / ((*uSyncPropPhase1Seg* + *uPhase2Seg* + 1) $*$ (*uQuantumPrescaler*))

This means that with *uSyncPropPhase1Seg* = 4, *uPhase2Seg* = 1, *uQuantumPrescaler* = 2 and an 8 MHz CAN clock, that the bit rate will be (8 MHz) / ((5 + 2 + 1) $*$ 2) or 500 Kbit/sec.

Returns:
None.

3.2.5.4 CANClkSourceSelect

Select CAN peripheral clock source

Prototype:
```
void
CANClkSourceSelect(uint32_t ui32Base,
                   uint16_t ui16Source)
```

Parameters:
ui32Base is the base address of the CAN controller to disable.
ui16Source is the clock source to select for the given CAN peripheral:
0 - Selected CPU SYSCLKOUT (CPU1.Sysclk or CPU2.Sysclk) (default at reset)
1 - External Oscillator (OSC) clock (direct from X1/X2)
2 - AUXCLKIN = GPIOn(GPIO19)

Description:
Selects the desired clock source for use with a given CAN peripheral.

Returns:
None.

3.2.5.5 CANDisable

Disables the CAN controller.

Prototype:
```
void
CANDisable(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the CAN controller to disable.

Description:
Disables the CAN controller for message processing. When disabled, the controller will no longer automatically process data on the CAN bus. The controller can be restarted by calling CANEnable(). The state of the CAN controller and the message objects in the controller are left as they were before this call was made.

Returns:
None.

3.2.5.6 CANEnable

Enables the CAN controller.

Prototype:

```
void
CANEnable(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the CAN controller to enable.

Description:

Enables the CAN controller for message processing. Once enabled, the controller will automatically transmit any pending frames, and process any received frames. The controller can be stopped by calling CANDisable(). Prior to calling CANEnable(), CANInit() should have been called to initialize the controller and the CAN bus clock should be configured by calling CANBitTimingSet().

Returns:

None.

3.2.5.7 CANErrCntrGet

Reads the CAN controller error counter register.

Prototype:

```
bool
CANErrCntrGet(uint32_t ui32Base,
              uint32_t *pui32RxCount,
              uint32_t *pui32TxCount)
```

Parameters:

ui32Base is the base address of the CAN controller.

pui32RxCount is a pointer to storage for the receive error counter.

pui32TxCount is a pointer to storage for the transmit error counter.

Description:

Reads the error counter register and returns the transmit and receive error counts to the caller along with a flag indicating if the controller receive counter has reached the error passive limit. The values of the receive and transmit error counters are returned through the pointers provided as parameters.

After this call, $*$*pui32RxCount* will hold the current receive error count and $*$*pui32TxCount* will hold the current transmit error count.

Returns:

Returns **true** if the receive error count has reached the error passive limit, and **false** if the error count is below the error passive limit.

3.2.5.8 CANGlobalIntClear

CAN Global interrupt Clear function.

Prototype:

```
void
CANGlobalIntClear(uint32_t ui32Base,
                  uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

Clear the specific CAN interrupt bit in the global interrupt flag register.

The *ui32IntFlags* parameter is the logical OR of any of the following:

CAN_GLB_INT_CANINT0 -Global Interrupt bit for CAN INT0 CAN_GLB_INT_CANINT1 -Global Interrupt bit for CAN INT1

Returns:

None.

3.2.5.9 CANGlobalIntDisable

CAN Global interrupt Disable function.

Prototype:

```
void
CANGlobalIntDisable(uint32_t ui32Base,
                    uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

Disables the specific CAN interrupt in the global interrupt enable register

The *ui32IntFlags* parameter is the logical OR of any of the following:

CAN_GLB_INT_CANINT0 -Global Interrupt bit for CAN INT0 CAN_GLB_INT_CANINT1 -Global Interrupt bit for CAN INT1

Returns:

None.

3.2.5.10 CANGlobalIntEnable

CAN Global interrupt Enable function.

Prototype:

```
void
CANGlobalIntEnable(uint32_t ui32Base,
                   uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:
Enables specific CAN interrupt in the global interrupt enable register

The *ui32IntFlags* parameter is the logical OR of any of the following:

CAN_GLB_INT_CANINT0 -Global Interrupt Enable bit for CAN INT0 CAN_GLB_INT_CANINT1 -Global Interrupt Enable bit for CAN INT1

Returns:
None.

3.2.5.11 CANGlobalIntstatusGet

CAN Global interrupt Status function.

Prototype:
```
bool
CANGlobalIntstatusGet(uint32_t ui32Base,
                      uint32_t ui32IntFlags)
```

Parameters:
ui32Base is the base address of the CAN controller.
ui32IntFlags is the bit mask of the interrupt sources to be checked.

Description:
Get the status of the specific CAN interrupt bits in the global interrupt flag register.

The *ui32IntFlags* parameter is the logical OR of any of the following:

CAN_GLB_INT_CANINT0 -Global Interrupt bit for CAN INT0 CAN_GLB_INT_CANINT1 -Global Interrupt bit for CAN INT1

Returns:
True if any of the requested interrupt bit(s) is (are) set.

3.2.5.12 CANInit

Initializes the CAN controller after reset.

Prototype:
```
void
CANInit(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the CAN controller.

Description:
After reset, the CAN controller is left in the disabled state. However, the memory used for message objects contains undefined values and must be cleared prior to enabling the CAN controller the first time. This prevents unwanted transmission or reception of data before the message objects are configured. This function must be called before enabling the controller the first time.

Returns:
None.

3.2.5.13 CANIntClear

Clears a CAN interrupt source.

Prototype:

```
void
CANIntClear(uint32_t ui32Base,
            uint32_t ui32IntClr)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32IntClr is a value indicating which interrupt source to clear.

Description:

This function can be used to clear a specific interrupt source. The *ui32IntClr* parameter should be one of the following values:

- **CAN_INT_INTID_STATUS** - Clears a status interrupt.
- 1-32 - Clears the specified message object interrupt

It is not necessary to use this function to clear an interrupt. This should only be used if the application wants to clear an interrupt source without taking the normal interrupt action.

Normally, the status interrupt is cleared by reading the controller status using CANStatusGet(). A specific message object interrupt is normally cleared by reading the message object using CANMessageGet().

Note:

Since 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 (since NVIC still sees the interrupt source asserted).

Returns:

None.

3.2.5.14 CANIntDisable

Disables individual CAN controller interrupt sources.

Prototype:

```
void
CANIntDisable(uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:

Disables the specified CAN controller interrupt sources. Only enabled interrupt sources can cause a processor interrupt.

The *ui32IntFlags* parameter has the same definition as in the CANIntEnable() function.

Returns:

None.

3.2.5.15 CANIntEnable

Enables individual CAN controller interrupt sources.

Prototype:

```
void
CANIntEnable(uint32_t ui32Base,
             uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

Enables specific interrupt sources of the CAN controller. Only enabled sources will cause a processor interrupt.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **CAN_INT_ERROR** - a controller error condition has occurred
- **CAN_INT_STATUS** - a message transfer has completed, or a bus error has been detected
- **CAN_INT_IE0** - allow CAN controller to generate interrupts on interrupt line 0
- **CAN_INT_IE1** - allow CAN controller to generate interrupts on interrupt line 1

In order to generate status or error interrupts, **CAN_INT_IE0** must be enabled. Further, for any particular transaction from a message object to generate an interrupt, that message object must have interrupts enabled (see CANMessageSet()). **CAN_INT_ERROR** will generate an interrupt if the controller enters the “bus off” condition, or if the error counters reach a limit. **CAN_INT_STATUS** will generate an interrupt under quite a few status conditions and may provide more interrupts than the application needs to handle. When an interrupt occurs, use CANIntStatus() to determine the cause.

Returns:

None.

3.2.5.16 CANIntRegister

Registers an interrupt handler for the CAN controller.

Prototype:

```
void
CANIntRegister(uint32_t ui32Base,
               unsigned char ucIntNumber,
               void (*pfnHandler)(void))
```

Parameters:

ui32Base is the base address of the CAN controller.

ucIntNumber is the interrupt line to register (0 or 1).

pfnHandler is a pointer to the function to be called when the enabled CAN interrupts occur.

Description:

This function registers the interrupt handler in the interrupt vector table, and enables CAN interrupts on the interrupt controller; specific CAN interrupt sources must be enabled using CANIntEnable(). The interrupt handler being registered must clear the source of the interrupt using CANIntClear().

If the application is using a static interrupt vector table stored in flash, then it is not necessary to register the interrupt handler this way. Instead, IntEnable() should be used to enable CAN interrupts on the interrupt controller.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

3.2.5.17 CANIntStatus

Returns the current CAN controller interrupt status.

Prototype:

```
uint32_t
CANIntStatus(uint32_t ui32Base,
             tCANIntStsReg eIntStsReg)
```

Parameters:

ui32Base is the base address of the CAN controller.

eIntStsReg indicates which interrupt status register to read

Description:

Returns the value of one of two interrupt status registers. The interrupt status register read is determined by the *eIntStsReg* parameter, which can have one of the following values:

- **CAN_INT_STS_CAUSE** - indicates the cause of the interrupt
- **CAN_INT_STS_OBJECT** - indicates pending interrupts of all message objects

CAN_INT_STS_CAUSE returns the value of the controller interrupt register and indicates the cause of the interrupt. It will be a value of **CAN_INT_INT0ID_STATUS** if the cause is a status interrupt. In this case, the status register should be read with the CANStatusGet() function. Calling this function to read the status will also clear the status interrupt. If the value of the interrupt register is in the range 1-32, then this indicates the number of the highest priority message object that has an interrupt pending. The message object interrupt can be cleared by using the CANIntClear() function, or by reading the message using CANMessageGet() in the case of a received message. The interrupt handler can read the interrupt status again to make sure all pending interrupts are cleared before returning from the interrupt.

CAN_INT_STS_OBJECT returns a bit mask indicating which message objects have pending interrupts. This can be used to discover all of the pending interrupts at once, as opposed to repeatedly reading the interrupt register by using **CAN_INT_STS_CAUSE**.

Returns:
Returns the value of one of the interrupt status registers.

3.2.5.18 CANIntUnregister

Unregisters an interrupt handler for the CAN controller.

Prototype:
```
void
CANIntUnregister(uint32_t ui32Base,
                 unsigned char ucIntNumber)
```

Parameters:
ui32Base is the base address of the controller.
ucIntNumber is the interrupt line to un-register (0 or 1).

Description:
This function unregisters the previously registered interrupt handler and disables the interrupt on the interrupt controller.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

3.2.5.19 CANMessageClear

Clears a message object so that it is no longer used.

Prototype:
```
void
CANMessageClear(uint32_t ui32Base,
                uint32_t ui32ObjID)
```

Parameters:
ui32Base is the base address of the CAN controller.
ui32ObjID is the message object number to disable (1-32).

Description:
This function frees the specified message object from use. Once a message object has been “cleared, ” it will no longer automatically send or receive messages, or generate interrupts.

Returns:
None.

3.2.5.20 CANMessageGet

Reads a CAN message from one of the message object buffers.

Prototype:

```
void
CANMessageGet(uint32_t ui32Base,
              uint32_t ui32ObjID,
              tCANMsgObject *pMsgObject,
              bool bClrPendingInt)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32ObjID is the object number to read (1-32).

pMsgObject points to a structure containing message object fields.

bClrPendingInt indicates whether an associated interrupt should be cleared.

Description:

This function is used to read the contents of one of the 32 message objects in the CAN controller, and return it to the caller. The data returned is stored in the fields of the caller-supplied structure pointed to by *pMsgObject*. The data consists of all of the parts of a CAN message, plus some control and status information.

Normally this is used to read a message object that has received and stored a CAN message with a certain identifier. However, this could also be used to read the contents of a message object in order to load the fields of the structure in case only part of the structure needs to be changed from a previous setting.

When using CANMessageGet, all of the same fields of the structure are populated in the same way as when the CANMessageSet() function is used, with the following exceptions:

pMsgObject->ui32Flags:

- **MSG_OBJ_NEW_DATA** indicates if this is new data since the last time it was read
- **MSG_OBJ_DATA_LOST** indicates that at least one message was received on this message object, and not read by the host before being overwritten.

Returns:

None.

3.2.5.21 CANMessageSet

Configures a message object in the CAN controller.

Prototype:

```
void
CANMessageSet(uint32_t ui32Base,
              uint32_t ui32ObjID,
              tCANMsgObject *pMsgObject,
              tMsgObjType eMsgType)
```

Parameters:

ui32Base is the base address of the CAN controller.

ui32ObjID is the object number to configure (1-32).

pMsgObject is a pointer to a structure containing message object settings.

eMsgType indicates the type of message for this object.

Description:

This function is used to configure any one of the 32 message objects in the CAN controller. A message object can be configured as any type of CAN message object as well as several options for automatic transmission and reception. This call also allows the message object to be configured to generate interrupts on completion of message receipt or transmission. The message object can also be configured with a filter/mask so that actions are only taken when a message that meets certain parameters is seen on the CAN bus.

The *eMsgType* parameter must be one of the following values:

- **MSG_OBJ_TYPE_TX** - CAN transmit message object.
- **MSG_OBJ_TYPE_TX_REMOTE** - CAN transmit remote request message object.
- **MSG_OBJ_TYPE_RX** - CAN receive message object.
- **MSG_OBJ_TYPE_RX_REMOTE** - CAN receive remote request message object.
- **MSG_OBJ_TYPE_RXTX_REMOTE** - CAN remote frame receive remote, then transmit message object.

The message object pointed to by *pMsgObject* must be populated by the caller, as follows:

- *ui32MsgID* - contains the message ID, either 11 or 29 bits.
- *ui32MsgIDMask* - mask of bits from *ui32MsgID* that must match if identifier filtering is enabled.
- *ui32Flags*
  - Set **MSG_OBJ_TX_INT_ENABLE** flag to enable interrupt on transmission.
  - Set **MSG_OBJ_RX_INT_ENABLE** flag to enable interrupt on receipt.
  - Set **MSG_OBJ_USE_ID_FILTER** flag to enable filtering based on the identifier mask specified by *ui32MsgIDMask*.
- *ui32MsgLen* - the number of bytes in the message data. This should be non-zero even for a remote frame; it should match the expected bytes of the data responding data frame.
- *pucMsgData* - points to a buffer containing up to 8 bytes of data for a data frame.

Example: To send a data frame or remote frame(in response to a remote request), take the following steps:

1. Set *eMsgType* to **MSG_OBJ_TYPE_TX**.
2. Set *pMsgObject->ui32MsgID* to the message ID.
3. Set *pMsgObject->ui32Flags*. Make sure to set **MSG_OBJ_TX_INT_ENABLE** to allow an interrupt to be generated when the message is sent.
4. Set *pMsgObject->ui32MsgLen* to the number of bytes in the data frame.
5. Set *pMsgObject->pucMsgData* to point to an array containing the bytes to send in the message.
6. Call this function with *ui32ObjID* set to one of the 32 object buffers.

Example: To receive a specific data frame, take the following steps:

1. Set *eMsgObjType* to **MSG_OBJ_TYPE_RX**.
2. Set *pMsgObject->ui32MsgID* to the full message ID, or a partial mask to use partial ID matching.

3. Set *pMsgObject->ui32MsgIDMask* bits that should be used for masking during comparison.
4. Set *pMsgObject->ui32Flags* as follows:
   - Set **MSG_OBJ_TX_INT_ENABLE** flag to be interrupted when the data frame is received.
   - Set **MSG_OBJ_USE_ID_FILTER** flag to enable identifier based filtering.
5. Set *pMsgObject->ui32MsgLen* to the number of bytes in the expected data frame.
6. The buffer pointed to by *pMsgObject->pucMsgData* and *pMsgObject->ui32MsgLen* are not used by this call as no data is present at the time of the call.
7. Call this function with *ui32ObjID* set to one of the 32 object buffers.

If you specify a message object buffer that already contains a message definition, it will be overwritten.

Returns:
None.

3.2.5.22 CANRetryGet

Returns the current setting for automatic retransmission.

Prototype:
```
bool
CANRetryGet(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the CAN controller.

Description:
Reads the current setting for the automatic retransmission in the CAN controller and returns it to the caller.

Returns:
Returns **true** if automatic retransmission is enabled, **false** otherwise.

3.2.5.23 CANRetrySet

Sets the CAN controller automatic retransmission behavior.

Prototype:
```
void
CANRetrySet(uint32_t ui32Base,
            bool bAutoRetry)
```

Parameters:
ui32Base is the base address of the CAN controller.
bAutoRetry enables automatic retransmission.

Description:
Enables or disables automatic retransmission of messages with detected errors. If *bAutoRetry* is **true**, then automatic retransmission is enabled, otherwise it is disabled.

Returns:
None.

3.2.5.24 CANStatusGet

Reads one of the controller status registers.

Prototype:
```
uint32_t
CANStatusGet(uint32_t ui32Base,
             tCANStsReg eStatusReg)
```

Parameters:
ui32Base is the base address of the CAN controller.
eStatusReg is the status register to read.

Description:
Reads a status register of the CAN controller and returns it to the caller. The different status registers are:

- **CAN_STS_CONTROL** - the main controller status
- **CAN_STS_TXREQUEST** - bit mask of objects pending transmission
- **CAN_STS_NEWDAT** - bit mask of objects with new data
- **CAN_STS_MSGVAL** - bit mask of objects with valid configuration

When reading the main controller status register, a pending status interrupt will be cleared. This should be used in the interrupt handler for the CAN controller if the cause is a status interrupt. The controller status register fields are as follows:

- **CAN_STATUS_PDA** - controller in local power down mode
- **CAN_STATUS_WAKE_UP** - controller initiated system wake up
- **CAN_STATUS_PERR** - parity error detected
- **CAN_STATUS_BUS_OFF** - controller is in bus-off condition
- **CAN_STATUS_EWARN** - an error counter has reached a limit of at least 96
- **CAN_STATUS_EPASS** - CAN controller is in the error passive state
- **CAN_STATUS_RXOK** - a message was received successfully (independent of any message filtering).
- **CAN_STATUS_TXOK** - a message was successfully transmitted
- **CAN_STATUS_LEC_NONE** - no error
- **CAN_STATUS_LEC_STUFF** - stuffing error detected
- **CAN_STATUS_LEC_FORM** - a format error occurred in the fixed format part of a message
- **CAN_STATUS_LEC_ACK** - a transmitted message was not acknowledged
- **CAN_STATUS_LEC_BIT1** - dominant level detected when trying to send in recessive mode
- **CAN_STATUS_LEC_BIT0** - recessive level detected when trying to send in dominant mode
- **CAN_STATUS_LEC_CRC** - CRC error in received message

The remaining status registers are 32-bit bit maps to the message objects. They can be used to quickly obtain information about the status of all the message objects without needing to query each one. They contain the following information:

- **CAN_STS_TXREQUEST** - if a message object's TxRequest bit is set, that means that a transmission is pending on that object. The application can use this to determine which objects are still waiting to send a message.
- **CAN_STS_NEWDAT** - if a message object's NewDat bit is set, that means that a new message has been received in that object, and has not yet been picked up by the host application
- **CAN_STS_MSGVAL** - if a message object's MsgVal bit is set, that means it has a valid configuration programmed. The host application can use this to determine which message objects are empty/unused.

Returns:

Returns the value of the status register.

3.3 CAN Message Objects

This section will explains how to configure the CAN message objects in various modes using the CANMessageSet() and CANMessageGet() APIs. The configuration of a message object is determined by two parameters that are passed into the CANMessageSet() API. These are the tCANMsgObject structure and the tMsgObjType type field. It is important to note that the ulObjID parameter is the index of one of the 32 message objects that are available and is not the message object's identifier.

Message objects can be defined as one of five types based on the needs of the application. They are defined in the tMsgObjType enumeration and can only be one of those values. The simplest of the message object types are MSG_OBJ_TYPE_TX and MSG_OBJ_TYPE_RX which are used to send or receive messages for a given message identifier or a range of identifiers. The message type MSG_OBJ_TYPE_TX_REMOTE is used to transmit a remote request for data from another CAN node on the network. These message objects do not transmit any data but once they send the request will automatically turn into receive message object and wait for data from a remote CAN device. The message type MSG_OBJ_TYPE_RX_REMOTE is the receiving end of a remote request, and receive remote requests for data and generate an interrupt to let the application know when to supply and transmit data back to the CAN controller that issued the remote request for data. The message type MSG_OBJ_TYPE_RXTX_REMOTE is similar to the MSG_OBJ_TYPE_RX_REMOTE except that it automatically responds with data that the application placed in the message object.

The remaining information used to configure a CAN message object is contained in the tCANMsgObject structure which is used when calling CANMessageSet() or will be filled by data read from the message object when calling CANMessageGet(). The CAN message identifier is simply stored into the ulMsgID member of the tCANMsgObject structure and is the 11 or 20 bit CAN identifier for this message object. The ulMsgIDMask is the mask is used in combination with the ulMsgID value to determine a match when the MSG_OBJ_USE_ID_FILTER flag is set for a message object. The ulMsgIDMask is ignored if MSG_OBJ_USE_ID_FILTER flag is not set. The last of the configuration parameters are specified in the ulFlags which are defined as a combination of the MSG_OBJ_$*$ values. The MSG_OBJ_TX_INT_ENABLE and MSG_OBJ_RX_INT_ENABLE flags will enable transmit complete or receive data interrupts. If the CAN network is only using extended(20 bit) identifiers then the MSG_OBJ_EXTENDED_ID flag should be specified. The CANMessageSet() function will force this flag set if the identifier is greater than an 11 bit identifier can hold. The MSG_OBJ_USE_ID_FILTER is used to enable filtering based on the message identifiers as message are seen by the CAN controller. The combination of ulMsgID and ulMsgIDMask will determine if a message is accepted for a given message object. In some cases it may be necessary to add a

filter based on the direction of the message, so in these cases the MSG_OBJ_USE_DIR_FILTER is used to only accept the direction specified in the message type. Another additional filter flag is MSG_OBJ_USE_EXT_FILTER which will filter on only extended identifiers. In a mixed 11 bit and 20 bit identifier system, this will prevent an 11 bit identifier being confused with a 20 bit identifier of the same value. It is not necessary to specify this if there are only extended identifiers being used in the system. To determine if the incoming message identifier matches a given message object, the incoming message identifier is ANDed with ulMsgIDMask and compared with ulMsgID. The "C" logic would be the following:

```
if((IncomingID & ulMsgIDMask) == ulMsgID)
{
    // Accept the message.
}
else
{
    // Ignore the message.
}
```

The last of the flags to affect CANMessageSet() is the MSG_OBJ_FIFO flag. This flag is used when combining multiple message objects in a FIFO. This is useful when an application needs to receive more than the 8 bytes of data that can be received by a single CAN message object. It can also be used to reduce the likelihood of causing an overrun of data on a single message object that may be receiving data faster than the application can handle when using a single message object. If multiple message objects are going to be used in a FIFO they must be read in sequential order based on the message object number and have the exact same message identifiers and filtering values. All but the last of the message objects in a FIFO should have the MSG_OBJ_FIFO and the last message object in the FIFO should not have the MSG_OBJ_FIFO flag set to specify that is the last entry in the FIFO. See the CAN FIFO configuration example in the Programming Examples section of this document.

The remaining flags are all used when calling CANMessageGet() when reading data or checking the status of a message object. If the MSG_OBJ_NEW_DATA flag is set in the tCANMsgObject ulFlags variable then the data returned was new and not stale data from a previous call to CANMessageGet(). If the MSG_OBJ_DATA_LOST flag is set then data was lost since this message object was last read with CANMessageGet(). The MSG_OBJ_REMOTE_FRAME flag will be set if the message object was configured as a remote message object and a remote request was received.

When sending or receiving data, the last two variables define the size and a pointer to the data used by CANMessageGet() and CANMessageSet(). The ulMsgLen variable in tCANMsgObject specifies the number of bytes to send when calling CANMessageSet() and the number of bytes to read when calling CANMessageGet(). The pucMsgData variable in tCANMsgObject is the pointer to the data to send ulMsgLen bytes, or the pointer to the buffer to read ulMsgLen bytes into.

3.4 Programming Examples

This example code will send out data from CAN controller 0 to be received by CAN controller 1. In order to actually receive the data, an external cable must be connected between the two ports. In this example, both controllers are configured for 1 Mbit operation.

```
    tCANBitClkParms CANBitClk;
```

```
tCANMsgObject sMsgObjectRx;
tCANMsgObject sMsgObjectTx;
unsigned char ucBufferIn[8];
unsigned char ucBufferOut[8];

// Reset the state of all the message objects and the state of the CAN
// module to a known state.
CANInit(CAN0_BASE);
CANInit(CAN1_BASE);

// Configure the controller for 1 Mbit operation.
CANSetBitTiming(CAN1_BASE, &CANBitClk);

// Take the CAN0 device out of INIT state.
CANEnable(CAN0_BASE);
CANEnable(CAN1_BASE);

// Configure a receive object.
sMsgObjectRx.ulMsgID = (0x400);
sMsgObjectRx.ulMsgIDMask = 0x7f8;
sMsgObjectRx.ulFlags = MSG_OBJ_USE_ID_FILTER | MSG_OBJ_FIFO;

// The first three message objects have the MSG_OBJ_FIFO set to
// indicate that they are part of a FIFO.
CANMessageSet(CAN0_BASE, 1, &sMsgObjectRx, MSG_OBJ_TYPE_RX);
CANMessageSet(CAN0_BASE, 2, &sMsgObjectRx, MSG_OBJ_TYPE_RX);
CANMessageSet(CAN0_BASE, 3, &sMsgObjectRx, MSG_OBJ_TYPE_RX);

// Last message object does not have the MSG_OBJ_FIFO set to
// indicate that this is the last message.
sMsgObjectRx.ulFlags = MSG_OBJ_USE_ID_FILTER;
CANMessageSet(CAN0_BASE, 4, &sMsgObjectRx, MSG_OBJ_TYPE_RX);

// Configure and start transmit of message object.
sMsgObjectTx.ulMsgID = 0x400;
sMsgObjectTx.ulFlags = 0;
sMsgObjectTx.ulMsgLen = 8;
sMsgObjectTx.pucMsgData = ucBufferOut;
CANMessageSet(CAN0_BASE, 2, &sMsgObjectTx, MSG_OBJ_TYPE_TX);

// Wait for new data to become available.
while((CANStatusGet(CAN1_BASE, CAN_STS_NEWDAT) & 1) == 0)
{
    // Read the message out of the message object.
    CANMessageGet(CAN1_BASE, 1, &sMsgObjectRx, true);
}

// Process new data in sMsgObjectRx.pucMsgData.
...
```

This example code will configure a set of CAN message objects in FIFO mode, using CAN controller 0.

```
tCANBitClkParms CANBitClk;
tCANMsgObject sMsgObjectRx;
unsigned char ucBufferIn[8];
unsigned char ucBufferOut[8];

// Reset the state of all the message objects and the state of
// the CAN module to a known state.
CANInit(CAN0_BASE);

// Configure the controller for 1 Mbit operation.
CANBitRateSet(CAN0_BASE, 8000000, 1000000);

// Take the CAN0 device out of INIT state.
CANEnable(CAN0_BASE);

// Configure a receive object this CAN FIFO to receive message
// objects with message ID 0x400-0x407.
sMsgObjectRx.ulMsgID = (0x400);
sMsgObjectRx.ulMsgIDMask = 0x7f8;
sMsgObjectRx.ulFlags = MSG_OBJ_USE_ID_FILTER | MSG_OBJ_FIFO;

// The first three message objects have the MSG_OBJ_FIFO set
// to indicate that they are part of a FIFO.
CANMessageSet(CAN0_BASE, 1, &sMsgObjectRx, MSG_OBJ_TYPE_RX);
CANMessageSet(CAN0_BASE, 2, &sMsgObjectRx, MSG_OBJ_TYPE_RX);
CANMessageSet(CAN0_BASE, 3, &sMsgObjectRx, MSG_OBJ_TYPE_RX);

// Last message object does not have the MSG_OBJ_FIFO set to
// indicate that this is the last message.
sMsgObjectRx.ulFlags = MSG_OBJ_USE_ID_FILTER;
CANMessageSet(CAN0_BASE, 4, &sMsgObjectRx, MSG_OBJ_TYPE_RX);

...
```

4 Interrupt Controller (PIE)

4.1 Introduction

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 0xD00 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.

4.2 API Functions

Functions

- void IntDisable (uint32_t ui32Interrupt)
- void IntEnable (uint32_t ui32Interrupt)
- bool IntMasterDisable (void)
- bool IntMasterEnable (void)
- void IntRegister (uint32_t ui32Interrupt, void (*pfnHandler)(void))
- void IntUnregister (uint32_t ui32Interrupt)

4.2.1 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.

4.2.2 Function Documentation

4.2.2.1 IntDisable

Disables an interrupt.

Prototype:

```
void
IntDisable(uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt to be disabled.

Description:

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.

Returns:

None.

4.2.2.2 IntEnable

Enables an interrupt.

Prototype:

```
void
IntEnable(uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt to be enabled.

Description:

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.

4.2.2.3 IntMasterDisable

Disables the processor interrupt.

Prototype:
```
bool
IntMasterDisable(void)
```

Description:
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 `bool`, 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.

4.2.2.4 IntMasterEnable

Enables the processor interrupt.

Prototype:
```
bool
IntMasterEnable(void)
```

Description:
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 `bool`, 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.

4.2.2.5 IntRegister

Registers a function to be called when an interrupt occurs.

Prototype:

```
void
IntRegister(uint32_t ui32Interrupt,
            void (*pfnHandler)(void))
```

Description:
Assumes PIE is enabled

Parameters:
ui32Interrupt 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 pre-empt other code, care must be taken to protect memory or peripherals that are accessed by the handler and other non-handler code.

Returns:
None.

4.2.2.6 IntUnregister

Unregisters the function to be called when an interrupt occurs.

Prototype:

```
void
IntUnregister(uint32_t ui32Interrupt)
```

Parameters:
ui32Interrupt specifies the interrupt in question.

Description:
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.

4.3 Programming Example

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();
```

5 System Control

5.1 Introduction

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

The members of the Delfino family have a varying peripheral set and memory sizes. The device has a set of read-only registers that indicate the size of the memories and the peripherals that are present. This information can be used to write adaptive software that will run on more than one member of the Delfino family.

The device can be clocked from one of four sources: an external oscillator, one of the two internal oscillators, or an external single ended clock source. The main PLL can be used with any of the three oscillators as its input, excluding the external single ended clock source.

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

5.2 API Functions

Defines

- SYSTEM_CLOCK_SPEED

Functions

- __asm (" .def _SysCtlDelay\n"" .sect \"ramfuncs\"\n"" .global _SysCtlDelay\n""_SysCtlDelay:\n"" SUB ACC, #1\n"" BF _SysCtlDelay, GEQ\n"" LRETR\n")
- void SysCtlAuxClockSet (uint32_t ui32Config)
- uint32_t SysCtlClockGet (uint32_t u32ClockIn)
- void SysCtlClockSet (uint32_t ui32Config)
- uint32_t SysCtlLowSpeedClockGet (uint32_t u32ClockIn)
- void SysCtlPeripheralDisable (uint32_t ui32Peripheral)
- void SysCtlPeripheralEnable (uint32_t ui32Peripheral)
- bool SysCtlPeripheralPresent (uint32_t ui32Peripheral)
- bool SysCtlPeripheralReady (uint32_t ui32Peripheral)
- void SysCtlPeripheralReset (uint32_t ui32Peripheral)
- void SysCtlReset (void)
- void SysCtlUSBPLLDisable (void)
- void SysCtlUSBPLLEnable (void)

5.2.1 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 SysCtlSRAMSizeGet(), SysCtlFlashSizeGet(), and SysCtlPeripheralPresent().

Clocking of the device is configured with SysCtlClockSet() and SysCtlAuxClockSet(). Information about device clocking is provided by SysCtlClockGet() and SysCtlLowSpeedClockGet().

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

5.2.2 Define Documentation

5.2.2.1 SYSTEM_CLOCK_SPEED

Definition:

```
#define SYSTEM_CLOCK_SPEED
```

Description:

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

5.2.3 Function Documentation

5.2.3.1 __asm

Provides a small delay.

Prototype:

```
__asm(".def _SysCtlDelay\n"".sect \"ramfuncs\"\n"".global
_SysCtlDelay\n""_SysCtlDelay:\n""SUB ACC,
      #1\n""BF _SysCtlDelay,
      GEQ\n""LRETR\n")
```

Parameters:

ulCount is the number of delay loop iterations to perform.

Description:

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 5 cycles/loop + 9.

Returns:

None.

5.2.3.2 SysCtlAuxClockSet

Sets the clocking of the device.

Prototype:

```
void
SysCtlAuxClockSet(uint32_t ui32Config)
```

Parameters:

ui32Config is the required configuration of the device clocking.

Description:

This function configures the clocking of the device. The input crystal frequency, oscillator to be used, use of the PLL, and the system clock divider are all configured with this function.

The *ui32Config* parameter is the logical OR of several different values, many of which are grouped into sets where only one can be chosen.

The system clock divider is chosen with one of the following values: **SYSCTL_SYSDIV_1**, **SYSCTL_SYSDIV_2**, **SYSCTL_SYSDIV_3**, ... **SYSCTL_SYSDIV_64**.

The use of the PLL is chosen with either **SYSCTL_USE_PLL** or **SYSCTL_USE_OSC**.

The external crystal frequency is chosen with one of the following values: **SYSCTL_XTAL_4MHZ**, **SYSCTL_XTAL_4_09MHZ**, **SYSCTL_XTAL_4_91MHZ**, **SYSCTL_XTAL_5MHZ**, **SYSCTL_XTAL_5_12MHZ**, **SYSCTL_XTAL_6MHZ**, **SYSCTL_XTAL_6_14MHZ**, **SYSCTL_XTAL_7_37MHZ**, **SYSCTL_XTAL_8MHZ**, **SYSCTL_XTAL_8_19MHZ**, **SYSCTL_XTAL_10MHZ**, **SYSCTL_XTAL_12MHZ**, **SYSCTL_XTAL_12_2MHZ**, **SYSCTL_XTAL_13_5MHZ**, **SYSCTL_XTAL_14_3MHZ**, **SYSCTL_XTAL_16MHZ**, **SYSCTL_XTAL_16_3MHZ**, **SYSCTL_XTAL_18MHZ**, **SYSCTL_XTAL_20MHZ**, **SYSCTL_XTAL_24MHZ**, or **SYSCTL_XTAL_25MHz**. Values below **SYSCTL_XTAL_5MHZ** are not valid when the PLL is in operation.

The oscillator source is chosen with one of the following values: **SYSCTL_OSC_MAIN**, **SYSCTL_OSC_INT**, **SYSCTL_OSC_INT4**, **SYSCTL_OSC_INT30**, or **SYSCTL_OSC_EXT32**. **SYSCTL_OSC_EXT32** is only available on devices with the hibernate module, and then only when the hibernate module has been enabled.

The internal and main oscillators are disabled with the **SYSCTL_INT_OSC_DIS** and **SYSCTL_MAIN_OSC_DIS** flags, respectively. The external oscillator must be enabled in order to use an external clock source. Note that attempts to disable the oscillator used to clock the device is prevented by the hardware.

To clock the system from an external source (such as an external crystal oscillator), use **SYSCTL_USE_OSC** | **SYSCTL_OSC_MAIN**. To clock the system from the main oscillator, use **SYSCTL_USE_OSC** | **SYSCTL_OSC_MAIN**. To clock the system from the PLL, use **SYSCTL_USE_PLL** | **SYSCTL_OSC_MAIN**, and select the appropriate crystal with one of the **SYSCTL_XTAL_xxx** values.

Note:

If selecting the PLL as the system clock source (that is, via **SYSCTL_USE_PLL**), this function polls the PLL lock interrupt to determine when the PLL has locked. If an interrupt handler for the system control interrupt is in place, and it responds to and clears the PLL lock interrupt, this function delays until its timeout has occurred instead of completing as soon as PLL lock is achieved.

Returns:

None.

5.2.3.3 SysCtlClockGet

Gets the processor clock rate.

Prototype:

```
uint32_t
SysCtlClockGet(uint32_t u32ClockIn)
```

Description:

This function determines the clock rate of the processor clock.

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.

5.2.3.4 SysCtlClockSet

Sets the clocking of the device.

Prototype:

```
void
SysCtlClockSet(uint32_t ui32Config)
```

Parameters:

ui32Config is the required configuration of the device clocking.

Description:

This function configures the clocking of the device. The oscillator to be used, SYSPLL fractional and integer multiplier, and the system clock divider are all configured with this function.

The *ui32Config* parameter is the logical OR of four values: Clock divider, Integer multiplier, Fractional multiplier, and oscillator source.

The system clock divider is chosen with using the following macro: **SYSCTL_SYSDIV(x)** - "x" is an integer of value 1 or any even value up to 126

The System PLL fractional multiplier is chosen with one of the following values: **SYSCTL_FMULT_0**, **SYSCTL_FMULT_1_4**, **SYSCTL_FMULT_1_2**, **SYSCTL_FMULT_3_4**

The System PLL integer multiplier is chosen with using the following macro: **SYSCTL_IMULT(x)** - "x" is an integer from 0 to 127

The oscillator source is chosen with one of the following values: **SYSCTL_OSCSRC_OSC2**, **SYSCTL_OSCSRC_XTAL**, **SYSCTL_OSCSRC_OSC1**

Note:

The external oscillator must be enabled in order to use an external clock source. Note that attempts to disable the oscillator used to clock the device is prevented by the hardware.

Returns:

None.

5.2.3.5 SysCtlLowSpeedClockGet

Gets the low speed peripheral clock rate.

Prototype:

```
uint32_t
SysCtlLowSpeedClockGet(uint32_t u32ClockIn)
```

Description:

This function determines the clock rate of the low speed peripherals.

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 low speed peripheral clock rate.

5.2.3.6 SysCtlPeripheralDisable

Disables a peripheral.

Prototype:

```
void
SysCtlPeripheralDisable(uint32_t ui32Peripheral)
```

Parameters:

ui32Peripheral is the peripheral to disable.

Description:

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

The *ui32Peripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_UART_A**, **SYSCTL_PERIPH_UART_B**, **SYSCTL_PERIPH_UART_C**, **SYSCTL_PERIPH_UART_D**, **SYSCTL_PERIPH_SPI_A**, **SYSCTL_PERIPH_SPI_B**, **SYSCTL_PERIPH_SPI_C**, **SYSCTL_PERIPH_MCBSP_A**, **SYSCTL_PERIPH_MCBSP_B**, **SYSCTL_PERIPH_DMA**, **SYSCTL_PERIPH_USB_A**

Returns:

None.

5.2.3.7 SysCtlPeripheralEnable

Enables a peripheral.

Prototype:

```
void
SysCtlPeripheralEnable(uint32_t ui32Peripheral)
```

Parameters:
ui32Peripheral is the peripheral to enable.

Description:
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 *ui32Peripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_UART_A**, **SYSCTL_PERIPH_UART_B**, **SYSCTL_PERIPH_UART_C**, **SYSCTL_PERIPH_UART_D**, **SYSCTL_PERIPH_SPI_A**, **SYSCTL_PERIPH_SPI_B**, **SYSCTL_PERIPH_SPI_C**, **SYSCTL_PERIPH_MCBSP_A**, **SYSCTL_PERIPH_MCBSP_B**, **SYSCTL_PERIPH_DMA**, **SYSCTL_PERIPH_USB_A**

Returns:
None.

5.2.3.8 SysCtlPeripheralPresent

Determines if a peripheral is present.

Prototype:
```
bool
SysCtlPeripheralPresent(uint32_t ui32Peripheral)
```

Parameters:
ui32Peripheral is the peripheral in question.

Description:
This function determines if a particular peripheral is present in the device. Each member of the family has a different peripheral set; this function determines which peripherals are present on this device.

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

5.2.3.9 SysCtlPeripheralReady

Determines if a peripheral is ready.

Prototype:
```
bool
SysCtlPeripheralReady(uint32_t ui32Peripheral)
```

Parameters:
ui32Peripheral is the peripheral in question.

Description:
This function determines if a particular peripheral is ready to be accessed. The peripheral may be in a non-ready state if it is not enabled, is being held in reset, or is in the process of becoming ready after being enabled or taken out of reset.

Note:
The ability to check for a peripheral being ready varies based on the part in use. Please consult the data sheet for the part you are using to determine if this feature is available.

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

5.2.3.10 SysCtlPeripheralReset

Resets a peripheral

Prototype:
```
void
SysCtlPeripheralReset(uint32_t ui32Peripheral)
```

Parameters:
ui32Peripheral is the peripheral to reset.

Description:
The f2837x devices do not have a means of resetting peripherals via via software. This is a dummy function that does nothing.

Returns:
None.

5.2.3.11 SysCtlReset

Resets the device.

Prototype:
```
void
SysCtlReset(void)
```

Description:
This function performs a software reset of the entire device. The processor and all peripherals are reset and all device registers are returned to their default values (with the exception of the reset cause register, which maintains its current value but has the software reset bit set as well).

Returns:
This function does not return.

5.2.3.12 SysCtlUSBPLLDisable

Powers down the USB PLL.

Prototype:
```
void
SysCtlUSBPLLDisable(void)
```

Description:
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.

5.2.3.13 SysCtlUSBPLLEnable

Powers up the USB PLL.

Prototype:
```
void
SysCtlUSBPLLEnable(void)
```

Description:
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.

5.3 Programming Example

The following example shows how to use the SysCtl API to enable device peripherals.

```
// Enable the EPWM1 and I2C1.
SysCtlPeripheralEnable(SYSCTL_PERIPH_EPWM1);
SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C1);
```

6 System Tick (SysTick)

6.1 Introduction

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.

6.2 API Functions

Functions

- void SysTickDisable (void)
- void SysTickEnable (void)
- void SysTickInit (void)
- void SysTickIntDisable (void)
- void SysTickIntEnable (void)
- void SysTickIntRegister (void (*pfnHandler)(void))
- void SysTickIntUnregister (void)
- uint32_t SysTickPeriodGet (void)
- void SysTickPeriodSet (uint32_t ui32Period)
- uint32_t SysTickValueGet (void)

6.2.1 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()).

6.2.2 Function Documentation

6.2.2.1 SysTickDisable

Disables the SysTick counter.

Prototype:

```
void
SysTickDisable(void)
```

Description:

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

Returns:

None.

6.2.2.2 SysTickEnable

Enables the SysTick counter.

Prototype:

```
void
SysTickEnable(void)
```

Description:

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.

6.2.2.3 SysTickInit

Initializes the Timer0 Module to act as a system tick

Prototype:

```
void
SysTickInit(void)
```

Returns:

None.

6.2.2.4 void SysTickIntDisable (void)

Disables the SysTick interrupt.

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

Returns:
None.

6.2.2.5 void SysTickIntEnable (void)

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.

6.2.2.6 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.

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

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

6.2.2.7 SysTickIntUnregister

Unregisters the interrupt handler for the SysTick interrupt.

Prototype:
```
void
SysTickIntUnregister(void)
```

Description:
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.

6.2.2.8 SysTickPeriodGet

Gets the period of the SysTick counter.

Prototype:

```
uint32_t
SysTickPeriodGet(void)
```

Description:

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.

6.2.2.9 SysTickPeriodSet

Sets the period of the SysTick counter.

Prototype:

```
void
SysTickPeriodSet(uint32_t ui32Period)
```

Parameters:

ui32Period is the number of clock ticks in each period of the SysTick counter; must be between 1 and 16, 777, 216, inclusive.

Description:

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 *ui32Period* supplied here on the next clock after the SysTick is enabled.

Returns:

None.

6.2.2.10 SysTickValueGet

Gets the current value of the SysTick counter.

Prototype:

```
uint32_t
SysTickValueGet(void)
```

Description:

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.

6.3 Programming Example

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);
SysTickEnable();

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

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

7 UART

7.1 Introduction

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

The Delfino UART 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.

Some of the features of the Delfino UART are:

- A 16x12 bit receive FIFO and a 16x8 bit transmit FIFO.
- Programmable baud rate generator.
- Automatic generation and stripping of start, stop, and parity bits.
- Line break generation and detection.
- Programmable serial interface
  - 5, 6, 7, or 8 data bits
  - even, odd, stick, or no parity bit generation and detection
  - 1 or 2 stop bit generation
  - baud rate generation, from DC to processor clock/16
- Modem control/flow control
- IrDA serial-IR (SIR) encoder/decoder.
- DMA interface

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

7.2 API Functions

Functions

- bool UARTBusy (uint32_t ui32Base)
- int32_t UARTCharGet (uint32_t ui32Base)
- int32_t UARTCharGetNonBlocking (uint32_t ui32Base)
- void UARTCharPut (uint32_t ui32Base, unsigned char ucData)
- bool UARTCharPutNonBlocking (uint32_t ui32Base, unsigned char ucData)
- bool UARTCharsAvail (uint32_t ui32Base)
- void UARTConfigGetExpClk (uint32_t ui32Base, uint32_t ui32UARTClk, uint32_t *pui32Baud, uint32_t *pui32Config)

- void UARTConfigSetExpClk (uint32_t ui32Base, uint32_t ui32UARTClk, uint32_t ui32Baud, uint32_t ui32Config)
- void UARTDisable (uint32_t ui32Base)
- void UARTEnable (uint32_t ui32Base)
- void UARTFIFODisable (uint32_t ui32Base)
- void UARTFIFOEnable (uint32_t ui32Base)
- void UARTFIFOIntLevelGet (uint32_t ui32Base, uint32_t *pui32TxLevel, uint32_t *pui32RxLevel)
- void UARTFIFOIntLevelSet (uint32_t ui32Base, uint32_t ui32TxLevel, uint32_t ui32RxLevel)
- void UARTFIFOLevelGet (uint32_t ui32Base, uint32_t *pui32TxLevel, uint32_t *pui32RxLevel)
- void UARTIntClear (uint32_t ui32Base, uint32_t ui32IntFlags)
- void UARTIntDisable (uint32_t ui32Base, uint32_t ui32IntFlags)
- void UARTIntEnable (uint32_t ui32Base, uint32_t ui32IntFlags)
- uint32_t UARTIntStatus (uint32_t ui32Base, bool bMasked)
- uint32_t UARTParityModeGet (uint32_t ui32Base)
- void UARTParityModeSet (uint32_t ui32Base, uint32_t ui32Parity)
- void UARTRxErrorClear (uint32_t ui32Base)
- uint32_t UARTRxErrorGet (uint32_t ui32Base)
- void UARTRXIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))
- void UARTRXIntUnregister (uint32_t ui32Base)
- void UARTsetLoopBack (uint32_t ui32Base, bool enable)
- bool UARTSpaceAvail (uint32_t ui32Base)
- uint32_t UARTTxIntModeGet (uint32_t ui32Base)
- void UARTTxIntModeSet (uint32_t ui32Base, uint32_t ui32Mode)
- void UARTTXIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))
- void UARTTXIntUnregister (uint32_t ui32Base)

7.2.1 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 UART ports on a Delfino 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. The DMA interface can be enabled or disabled by the UARTDMAEnable() and UARTDMADisable() 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.

7.2.2 Function Documentation

7.2.2.1 UARTBusy

Determines whether the UART transmitter is busy or not.

Prototype:

```
bool
UARTBusy(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

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.

Returns:

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

7.2.2.2 UARTCharGet

Waits for a character from the specified port.

Prototype:

```
int32_t
UARTCharGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

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*.

7.2.2.3 UARTCharGetNonBlocking

Receives a character from the specified port.

Prototype:

```
int32_t
UARTCharGetNonBlocking(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

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.

7.2.2.4 UARTCharPut

Waits to send a character from the specified port.

Prototype:

```
void
UARTCharPut(uint32_t ui32Base,
            unsigned char ucData)
```

Parameters:

ui32Base is the base address of the UART port.

ucData is the character to be transmitted.

Description:

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.

7.2.2.5 UARTCharPutNonBlocking

Sends a character to the specified port.

Prototype:

```
bool
UARTCharPutNonBlocking(uint32_t ui32Base,
                       unsigned char ucData)
```

Parameters:

ui32Base is the base address of the UART port.

ucData is the character to be transmitted.

Description:

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.

7.2.2.6 UARTCharsAvail

Determines if there are any characters in the receive FIFO.

Prototype:
```
bool
UARTCharsAvail(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the UART port.

Description:
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.

7.2.2.7 UARTConfigGetExpClk

Gets the current configuration of a UART.

Prototype:
```
void
UARTConfigGetExpClk(uint32_t ui32Base,
                    uint32_t ui32UARTClk,
                    uint32_t *pui32Baud,
                    uint32_t *pui32Config)
```

Parameters:
ui32Base is the base address of the UART port.
ui32UARTClk is the rate of the clock supplied to the UART module.
pui32Baud is a pointer to storage for the baud rate.
pui32Config is a pointer to storage for the data format.

Description:
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 *pui32Config* is enumerated the same as the *ui32Config* 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.

7.2.2.8 UARTConfigSetExpClk

Sets the configuration of a UART.

Prototype:
```
void
UARTConfigSetExpClk(uint32_t ui32Base,
                    uint32_t ui32UARTClk,
                    uint32_t ui32Baud,
                    uint32_t ui32Config)
```

Parameters:
ui32Base is the base address of the UART port.
ui32UARTClk is the rate of the clock supplied to the UART module.
ui32Baud is the desired baud rate.
ui32Config is the data format for the port (number of data bits, number of stop bits, and parity).

Description:
This function configures the UART for operation in the specified data format. The baud rate is provided in the *ui32Baud* parameter and the data format in the *ui32Config* parameter.

The *ui32Config* 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.

7.2.2.9 UARTDisable

Disables transmitting and receiving.

Prototype:
```
void
UARTDisable(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the UART port.

Description:
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.

7.2.2.10 UARTEnable

Enables transmitting and receiving.

Prototype:
```
void
UARTEnable(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the UART port.

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

Returns:
None.

7.2.2.11 UARTFIFODisable

Disables the transmit and receive FIFOs.

Prototype:
```
void
UARTFIFODisable(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the UART port.

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

Returns:
None.

7.2.2.12 UARTFIFOEnable

Enables the transmit and receive FIFOs.

Prototype:
```
void
UARTFIFOEnable(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the UART port.

Description:
This functions enables the transmit and receive FIFOs in the UART.

Returns:
None.

7.2.2.13 UARTFIFOIntLevelGet

Gets the FIFO level at which interrupts are generated.

Prototype:
```
void
UARTFIFOIntLevelGet(uint32_t ui32Base,
                    uint32_t *pui32TxLevel,
                    uint32_t *pui32RxLevel)
```

Parameters:
ui32Base is the base address of the UART port.
pui32TxLevel is a pointer to storage for the transmit FIFO level, returned as one of **UART_FIFO_TX1_8**, **UART_FIFO_TX2_8**, **UART_FIFO_TX4_8**, **UART_FIFO_TX6_8**, or **UART_FIFO_TX7_8**.
pui32RxLevel is a pointer to storage for the receive FIFO level, returned as one of **UART_FIFO_RX1_8**, **UART_FIFO_RX2_8**, **UART_FIFO_RX4_8**, **UART_FIFO_RX6_8**, or **UART_FIFO_RX7_8**.

Description:
This function gets the FIFO level at which transmit and receive interrupts are generated.

Returns:
None.

7.2.2.14 UARTFIFOIntLevelSet

Sets the FIFO level at which interrupts are generated.

Prototype:
```
void
UARTFIFOIntLevelSet(uint32_t ui32Base,
                    uint32_t ui32TxLevel,
                    uint32_t ui32RxLevel)
```

Parameters:
ui32Base is the base address of the UART port.
ui32TxLevel 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**.
ui32RxLevel 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**.

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

Returns:
None.

7.2.2.15 UARTFIFOLevelGet

Gets the FIFO level at which interrupts are generated.

Prototype:
```
void
UARTFIFOLevelGet(uint32_t ui32Base,
                 uint32_t *pui32TxLevel,
                 uint32_t *pui32RxLevel)
```

Parameters:
ui32Base is the base address of the UART port.
pui32TxLevel is a pointer to storage for the transmit FIFO level, returned as one of **UART_FIFO_TX1_8**, **UART_FIFO_TX2_8**, **UART_FIFO_TX4_8**, **UART_FIFO_TX6_8**, or **UART_FIFO_TX7_8**.
pui32RxLevel is a pointer to storage for the receive FIFO level, returned as one of **UART_FIFO_RX1_8**, **UART_FIFO_RX2_8**, **UART_FIFO_RX4_8**, **UART_FIFO_RX6_8**, or **UART_FIFO_RX7_8**.

Description:
This function gets the FIFO level at which transmit and receive interrupts are generated.

Returns:
None.

7.2.2.16 UARTIntClear

Clears UART interrupt sources.

Prototype:
```
void
UARTIntClear(uint32_t ui32Base,
             uint32_t ui32IntFlags)
```

Parameters:
ui32Base is the base address of the UART port.
ui32IntFlags is a bit mask of the interrupt sources to be cleared.

Description:
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 *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* 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.

7.2.2.17 UARTIntDisable

Disables individual UART interrupt sources.

Prototype:
```
void
UARTIntDisable(uint32_t ui32Base,
               uint32_t ui32IntFlags)
```

Parameters:
ui32Base is the base address of the UART port.
ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:
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 *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to UARTIntEnable().

Returns:
None.

7.2.2.18 UARTIntEnable

Enables individual UART interrupt sources.

Prototype:
```
void
UARTIntEnable(uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

Parameters:
ui32Base is the base address of the UART port.
ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:
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 *ui32IntFlags* 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.

7.2.2.19 UARTIntStatus

Gets the current interrupt status.

Prototype:
```
uint32_t
UARTIntStatus(uint32_t ui32Base,
              bool bMasked)
```

Parameters:
ui32Base 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.

Description:
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().

7.2.2.20 UARTParityModeGet

Gets the type of parity currently being used.

Prototype:
```
uint32_t
UARTParityModeGet(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the UART port.

Description:
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**.

7.2.2.21 UARTParityModeSet

Sets the type of parity.

Prototype:
```
void
UARTParityModeSet(uint32_t ui32Base,
                  uint32_t ui32Parity)
```

Parameters:
ui32Base is the base address of the UART port.
ui32Parity specifies the type of parity to use.

Description:
Sets the type of parity to use for transmitting and expect when receiving. The *ui32Parity* 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.

7.2.2.22 UARTRxErrorClear

Clears all reported receiver errors.

Prototype:
```
void
UARTRxErrorClear(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the UART port.

Description:
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.

7.2.2.23 UARTRxErrorGet

Gets current receiver errors.

Prototype:

```
uint32_t
UARTRxErrorGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

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**.

7.2.2.24 UARTRXIntRegister

Registers an interrupt handler for a UART RX interrupt.

Prototype:

```
void
UARTRXIntRegister(uint32_t ui32Base,
                  void (*pfnHandler)(void))
```

Parameters:

ui32Base is the base address of the UART port.

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

Description:

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.

7.2.2.25 UARTRXIntUnregister

Unregisters an interrupt handler for a UART RX interrupt.

Prototype:

```
void
UARTRXIntUnregister(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the UART port.

Description:
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.

7.2.2.26 UARTsetLoopBack

Enables Loop Back Test Mode.

Prototype:
```
void
UARTsetLoopBack(uint32_t ui32Base,
                bool enable)
```

Parameters:
ui32Base is the base address of the UART port.

Description:
Sets the SCICCR.LOOPBKENA to enable

Returns:
None.

7.2.2.27 UARTSpaceAvail

Determines if there is any space in the transmit FIFO.

Prototype:
```
bool
UARTSpaceAvail(uint32_t ui32Base)
```

Parameters:
ui32Base is the base address of the UART port.

Description:
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.

7.2.2.28 UARTTxIntModeGet

Returns the current operating mode for the UART transmit interrupt.

Prototype:

```
uint32_t
UARTTxIntModeGet(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

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**.

7.2.2.29 UARTTxIntModeSet

Sets the operating mode for the UART transmit interrupt.

Prototype:

```
void
UARTTxIntModeSet(uint32_t ui32Base,
                 uint32_t ui32Mode)
```

Parameters:

ui32Base is the base address of the UART port.

ui32Mode 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.

Description:

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 *ui32Mode* 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.

7.2.2.30 UARTTXIntRegister

Registers an interrupt handler for a UART TX interrupt.

Prototype:

```
void
UARTTXIntRegister(uint32_t ui32Base,
                  void (*pfnHandler)(void))
```

Parameters:

ui32Base is the base address of the UART port.

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

Description:

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.

7.2.2.31 UARTTXIntUnregister

Unregisters an interrupt handler for a UART TX interrupt.

Prototype:

```
void
UARTTXIntUnregister(uint32_t ui32Base)
```

Parameters:

ui32Base is the base address of the UART port.

Description:

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.

7.3 Programming Example

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);
```

8 USB Controller

8.1 Introduction

The USB APIs provide a set of functions that are used to access the Delfino USB device or host controllers. The APIs are split into groups according to the functionality provided by the USB controller present in the microcontroller. Because of this, the driver has to handle microcontrollers that have only a USB device interface, a host and/or device interface, or microcontrollers that have an OTG interface, The groups are the following: USBDev, USBHost, USBOTG, USBEndpoint, and USBFIFO. The APIs in the USBDev group are only used with microcontrollers that have a USB device controller. The APIs in the USBHost group can only be used with microcontrollers that have a USB host controller. The USBOTG APIs are used by microcontrollers with an OTG interface. With USB OTG controllers, once the mode of the USB controller is configured, the device or host APIs should be used. 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.

8.2 API Functions

Functions

- uint32_t USBDevAddrGet (uint32_t ui32Base)
- void USBDevAddrSet (uint32_t ui32Base, uint32_t ui32Address)
- void USBDevConnect (uint32_t ui32Base)
- void USBDevDisconnect (uint32_t ui32Base)
- void USBDevEndpointConfigGet (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t *pui32MaxPacketSize, uint32_t *pui32Flags)
- void USBDevEndpointConfigSet (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32MaxPacketSize, uint32_t ui32Flags)
- void USBDevEndpointDataAck (uint32_t ui32Base, uint32_t ui32Endpoint, bool bIsLastPacket)
- void USBDevEndpointStall (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void USBDevEndpointStallClear (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void USBDevEndpointStatusClear (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void USBDevMode (uint32_t ui32Base)
- uint32_t USBEndpointDataAvail (uint32_t ui32Base, uint32_t ui32Endpoint)
- int32_t USBEndpointDataGet (uint32_t ui32Base, uint32_t ui32Endpoint, uint8_t *pui8Data, uint32_t *pui32Size)

- int32_t USBEndpointDataPut (uint32_t ui32Base, uint32_t ui32Endpoint, uint8_t *pui8Data, uint32_t ui32Size)
- int32_t USBEndpointDataSend (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32TransType)
- void USBEndpointDataToggleClear (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void USBEndpointDMAChannel (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Channel)
- void USBEndpointDMAConfigSet (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Config)
- void USBEndpointDMADisable (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void USBEndpointDMAEnable (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void USBEndpointPacketCountSet (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Count)
- uint32_t USBEndpointStatus (uint32_t ui32Base, uint32_t ui32Endpoint)
- uint32_t USBFIFOAddrGet (uint32_t ui32Base, uint32_t ui32Endpoint)
- void USBFIFOConfigGet (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t *pui32FIFOAddress, uint32_t *pui32FIFOSize, uint32_t ui32Flags)
- void USBFIFOConfigSet (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32FIFOAddress, uint32_t ui32FIFOSize, uint32_t ui32Flags)
- void USBFIFOFlush (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- uint32_t USBFrameNumberGet (uint32_t ui32Base)
- uint32_t USBHostAddrGet (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void USBHostAddrSet (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Addr, uint32_t ui32Flags)
- void USBHostEndpointConfig (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32MaxPayload, uint32_t ui32NAKPollInterval, uint32_t ui32TargetEndpoint, uint32_t ui32Flags)
- void USBHostEndpointDataAck (uint32_t ui32Base, uint32_t ui32Endpoint)
- void USBHostEndpointDataToggle (uint32_t ui32Base, uint32_t ui32Endpoint, bool bDataToggle, uint32_t ui32Flags)
- void USBHostEndpointStatusClear (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- uint32_t USBHostHubAddrGet (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Flags)
- void USBHostHubAddrSet (uint32_t ui32Base, uint32_t ui32Endpoint, uint32_t ui32Addr, uint32_t ui32Flags)
- void USBHostMode (uint32_t ui32Base)
- void USBHostPwrConfig (uint32_t ui32Base, uint32_t ui32Flags)
- void USBHostPwrDisable (uint32_t ui32Base)
- void USBHostPwrEnable (uint32_t ui32Base)
- void USBHostPwrFaultDisable (uint32_t ui32Base)
- void USBHostPwrFaultEnable (uint32_t ui32Base)
- void USBHostRequestIN (uint32_t ui32Base, uint32_t ui32Endpoint)
- void USBHostRequestINClear (uint32_t ui32Base, uint32_t ui32Endpoint)
- void USBHostRequestStatus (uint32_t ui32Base)
- void USBHostReset (uint32_t ui32Base, bool bStart)
- void USBHostResume (uint32_t ui32Base, bool bStart)
- uint32_t USBHostSpeedGet (uint32_t ui32Base)

- void USBHostSuspend (uint32_t ui32Base)
- void USBIntDisableControl (uint32_t ui32Base, uint32_t ui32Flags)
- void USBIntDisableEndpoint (uint32_t ui32Base, uint32_t ui32Flags)
- void USBIntEnableControl (uint32_t ui32Base, uint32_t ui32Flags)
- void USBIntEnableEndpoint (uint32_t ui32Base, uint32_t ui32Flags)
- void USBIntRegister (uint32_t ui32Base, void (*pfnHandler)(void))
- uint32_t USBIntStatus (uint32_t ui32Base, uint32_t *pui32IntStatusEP)
- uint32_t USBIntStatusControl (uint32_t ui32Base)
- uint32_t USBIntStatusEndpoint (uint32_t ui32Base)
- void USBIntUnregister (uint32_t ui32Base)
- uint32_t USBModeGet (uint32_t ui32Base)
- uint32_t USBNumEndpointsGet (uint32_t ui32Base)
- void USBOTGMode (uint32_t ui32Base)
- void USBOTGSessionRequest (uint32_t ui32Base, bool bStart)
- void USBPHYPowerOff (uint32_t ui32Base)
- void USBPHYPowerOn (uint32_t ui32Base)

8.2.1 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. For microcontrollers with host controllers, the application can use an endpoint 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 FIFOs in devices that have a USB device controller as well as those that have a host controller. 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);
```

8.2.2 Function Documentation

8.2.2.1 USBDevAddrGet

Returns the current device address in device mode.

Prototype:

```
uint32_t
USBDevAddrGet(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

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

Note:

This function must only be called in device mode.

Returns:

The current device address.

8.2.2.2 USBDevAddrSet

Sets the address in device mode.

Prototype:

```
void
USBDevAddrSet(uint32_t ui32Base,
              uint32_t ui32Address)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Address is the address to use for a device.

Description:

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 must only be called in device mode.

Returns:
None.

8.2.2.3 USBDevConnect

Connects the USB controller to the bus in device mode.

Prototype:
```
void
USBDevConnect(uint32_t ui32Base)
```

Parameters:
ui32Base specifies the USB module base address.

Description:
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 must only be called in device mode.

Returns:
None.

8.2.2.4 USBDevDisconnect

Removes the USB controller from the bus in device mode.

Prototype:
```
void
USBDevDisconnect(uint32_t ui32Base)
```

Parameters:
ui32Base specifies the USB module base address.

Description:
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 must only be called in device mode.

Returns:
None.

8.2.2.5 USBDevEndpointConfigGet

Gets the current configuration for an endpoint.

Prototype:

```
void
USBDevEndpointConfigGet(uint32_t ui32Base,
                        uint32_t ui32Endpoint,
                        uint32_t *pui32MaxPacketSize,
                        uint32_t *pui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

pui32MaxPacketSize is a pointer which is written with the maximum packet size for this endpoint.

pui32Flags 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.

Description:

This function returns the basic configuration for an endpoint in device mode. The values returned in *∗pui32MaxPacketSize* and *∗pui32Flags* are equivalent to the *ui32MaxPacketSize* and *ui32Flags* previously passed to USBDevEndpointConfigSet() for this endpoint.

Note:

This function must only be called in device mode.

Returns:

None.

8.2.2.6 USBDevEndpointConfigSet

Sets the configuration for an endpoint.

Prototype:

```
void
USBDevEndpointConfigSet(uint32_t ui32Base,
                        uint32_t ui32Endpoint,
                        uint32_t ui32MaxPacketSize,
                        uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32MaxPacketSize is the maximum packet size for this endpoint.

ui32Flags are used to configure other endpoint settings.

Description:

This function sets the basic configuration for an endpoint in device mode. Endpoint zero does not have a dynamic configuration, so this function must not be called for endpoint zero. The

ui32Flags 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 *ui32MaxPacketSize* 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 *ui32MaxPacketSize* 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 must only be called in device mode.

Returns:
None.

8.2.2.7 USBDevEndpointDataAck

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

Prototype:

```
void
USBDevEndpointDataAck(uint32_t ui32Base,
                      uint32_t ui32Endpoint,
                      bool bIsLastPacket)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

bIsLastPacket indicates if this packet is the last one.

Description:

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 must only be called in device mode.

Returns:
None.

8.2.2.8 USBDevEndpointStall

Stalls the specified endpoint in device mode.

Prototype:
```
void
USBDevEndpointStall(uint32_t ui32Base,
                    uint32_t ui32Endpoint,
                    uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint specifies the endpoint to stall.
ui32Flags specifies whether to stall the IN or OUT endpoint.

Description:
This function causes the endpoint number passed in to go into a stall condition. If the *ui32Flags* parameter is **USB_EP_DEV_IN**, then the stall is issued on the IN portion of this endpoint. If the *ui32Flags* parameter is **USB_EP_DEV_OUT**, then the stall is issued on the OUT portion of this endpoint.

Note:
This function must only be called in device mode.

Returns:
None.

8.2.2.9 USBDevEndpointStallClear

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

Prototype:
```
void
USBDevEndpointStallClear(uint32_t ui32Base,
                         uint32_t ui32Endpoint,
                         uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint specifies which endpoint to remove the stall condition.
ui32Flags specifies whether to remove the stall condition from the IN or the OUT portion of this endpoint.

Description:
This function causes the endpoint number passed in to exit the stall condition. If the *ui32Flags* parameter is **USB_EP_DEV_IN**, then the stall is cleared on the IN portion of this endpoint. If the *ui32Flags* parameter is **USB_EP_DEV_OUT**, then the stall is cleared on the OUT portion of this endpoint.

Note:
This function must only be called in device mode.

Returns:
None.

8.2.2.10 USBDevEndpointStatusClear

Clears the status bits in this endpoint in device mode.

Prototype:
```
void
USBDevEndpointStatusClear(uint32_t ui32Base,
                          uint32_t ui32Endpoint,
                          uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
ui32Flags are the status bits that are cleared.

Description:
This function clears the status of any bits that are passed in the *ui32Flags* parameter. The *ui32Flags* parameter can take the value returned from the USBEndpointStatus() call.

Note:
This function must only be called in device mode.

Returns:
None.

8.2.2.11 USBDevMode

Change the mode of the USB controller to device.

Prototype:
```
void
USBDevMode(uint32_t ui32Base)
```

Parameters:
ui32Base specifies the USB module base address.

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

Note:
This function must only be called on microcontrollers that support OTG operation and have the DEVMODOTG bit in the USBGPCS register.

Returns:
None.

8.2.2.12 USBEndpointDataAvail

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

Prototype:
```
uint32_t
USBEndpointDataAvail(uint32_t ui32Base,
                     uint32_t ui32Endpoint)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.

Description:
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.

8.2.2.13 USBEndpointDataGet

Retrieves data from the given endpoint's FIFO.

Prototype:
```
int32_t
USBEndpointDataGet(uint32_t ui32Base,
                   uint32_t ui32Endpoint,
                   uint8_t *pui8Data,
                   uint32_t *pui32Size)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
pui8Data is a pointer to the data area used to return the data from the FIFO.
pui32Size is initially the size of the buffer passed into this call via the *pui8Data* parameter. It is set to the amount of data returned in the buffer.

Description:
This function returns the data from the FIFO for the given endpoint. The *pui32Size* parameter indicates the size of the buffer passed in the *pui32Data* parameter. The data in the *pui32Size* parameter is changed to match the amount of data returned in the *pui8Data* parameter. If a zero-byte packet is received, this call does not return an error but instead just returns a zero in the *pui32Size* 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.

8.2.2.14 USBEndpointDataPut

Puts data into the given endpoint's FIFO.

Prototype:
```
int32_t
USBEndpointDataPut(uint32_t ui32Base,
                   uint32_t ui32Endpoint,
                   uint8_t *pui8Data,
                   uint32_t ui32Size)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
pui8Data is a pointer to the data area used as the source for the data to put into the FIFO.
ui32Size is the amount of data to put into the FIFO.

Description:
This function puts the data from the *pui8Data* 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 must 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.

8.2.2.15 USBEndpointDataSend

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

Prototype:
```
int32_t
USBEndpointDataSend(uint32_t ui32Base,
                    uint32_t ui32Endpoint,
                    uint32_t ui32TransType)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
ui32TransType is set to indicate what type of data is being sent.

Description:
This function starts the transfer of data from the FIFO for a given endpoint. This function is called if the **USB_EP_AUTO_SET** bit was not enabled for the endpoint. Setting the *ui32TransType* parameter allows the appropriate signaling on the USB bus for the type of transaction being requested. The *ui32TransType* parameter must 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.

8.2.2.16 USBEndpointDataToggleClear

Sets the data toggle on an endpoint to zero.

Prototype:
```
void
USBEndpointDataToggleClear(uint32_t ui32Base,
                           uint32_t ui32Endpoint,
                           uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint specifies the endpoint to reset the data toggle.
ui32Flags specifies whether to access the IN or OUT endpoint.

Description:
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 *ui32Flags* parameter must be one of **USB_EP_HOST_OUT**, **USB_EP_HOST_IN**, **USB_EP_DEV_OUT**, or **USB_EP_DEV_IN**.

Returns:
None.

8.2.2.17 USBEndpointDMAChannel

Sets the DMA channel to use for a given endpoint.

Prototype:
```
void
USBEndpointDMAChannel(uint32_t ui32Base,
                      uint32_t ui32Endpoint,
                      uint32_t ui32Channel)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint specifies which endpoint's FIFO address to return.
ui32Channel specifies which DMA channel to use for which endpoint.

Description:

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 are passed into the *ui32Channel* 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.

8.2.2.18 USBEndpointDMAConfigSet

Configure the DMA settings for an endpoint.

Prototype:

```
void
USBEndpointDMAConfigSet(uint32_t ui32Base,
                        uint32_t ui32Endpoint,
                        uint32_t ui32Config)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Config specifies the configuration options for an endpoint.

Description:

This function configures the DMA settings for a given endpoint without changing other options that may already be configured. In order for the DMA transfer to be enabled, the USBEndpointDMAEnable() function must be called before starting the DMA transfer. The configuration options are passed in the *ui32Config* parameter and can have the values described below.

One of the following values to specify direction:

- **USB_EP_HOST_OUT** or **USB_EP_DEV_IN** - This setting is used with DMA transfers from memory to the USB controller.
- **USB_EP_HOST_IN** or **USB_EP_DEV_OUT** - This setting is used with DMA transfers from the USB controller to memory.

One of the following values:

- **USB_EP_DMA_MODE_0(default)** - This setting is typically used for transfers that do not span multiple packets or when interrupts are required for each packet.
- **USB_EP_DMA_MODE_1** - This setting is typically used for transfers that span multiple packets and do not require interrupts between packets.

Values only used with **USB_EP_HOST_OUT** or **USB_EP_DEV_IN:**

- **USB_EP_AUTO_SET** - This setting is used to allow transmit DMA transfers to automatically be sent when a full packet is loaded into a FIFO. This is needed with **USB_EP_DMA_MODE_1** to ensure that packets go out when the FIFO becomes full and the DMA has more data to send.

Values only used with **USB_EP_HOST_IN** or **USB_EP_DEV_OUT:**

- **USB_EP_AUTO_CLEAR** - This setting is used to allow receive DMA transfers to automatically be acknowledged as they are received. This is needed with **USB_EP_DMA_MODE_1** to ensure that packets continue to be received and acknowledged when the FIFO is emptied by the DMA transfer.

Values only used with **USB_EP_HOST_IN:**

- **USB_EP_AUTO_REQUEST** - This setting is used to allow receive DMA transfers to automatically request a new IN transaction when the previous transfer has emptied the FIFO. This is typically used in conjunction with **USB_EP_AUTO_CLEAR** so that receive DMA transfers can continue without interrupting the main processor.

Example: Set endpoint 1 receive endpoint to automatically acknowledge request and automatically generate a new IN request in host mode.

```
//
// Configure endpoint 1 for receiving multiple packets using DMA.
//
USBEndpointDMAConfigSet(USB0_BASE,  USB_EP_1,  USB_EP_HOST_IN |
                                               USB_EP_DMA_MODE_1 |
                                               USB_EP_AUTO_CLEAR |
                                               USB_EP_AUTO_REQUEST);
```

Example: Set endpoint 2 transmit endpoint to automatically send each packet in host mode when spanning multiple packets.

```
//
// Configure endpoint 1 for transmitting multiple packets using DMA.
//
USBEndpointDMAConfigSet(USB0_BASE,  USB_EP_2,  USB_EP_HOST_OUT |
                                               USB_EP_DMA_MODE_1 |
                                               USB_EP_AUTO_SET);
```

Returns:
None.

8.2.2.19 USBEndpointDMADisable

Disable DMA on a given endpoint.

Prototype:

```
void
USBEndpointDMADisable(uint32_t ui32Base,
                      uint32_t ui32Endpoint,
                      uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
ui32Flags specifies which direction to disable.

Description:
This function disables DMA on a given endpoint to allow non-DMA USB transactions to generate interrupts normally. The *ui32Flags* parameter must be **USB_EP_DEV_IN** or **USB_EP_DEV_OUT**; all other bits are ignored.

Returns:
None.

8.2.2.20 USBEndpointDMAEnable

Enable DMA on a given endpoint.

Prototype:
```
void
USBEndpointDMAEnable(uint32_t ui32Base,
                     uint32_t ui32Endpoint,
                     uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
ui32Flags specifies which direction and what mode to use when enabling DMA.

Description:
This function enables DMA on a given endpoint and configures the mode according to the values in the *ui32Flags* parameter. The *ui32Flags* parameter must have **USB_EP_DEV_IN** or **USB_EP_DEV_OUT** set. Once this function is called the only DMA or error interrupts are generated by the USB controller.

Note:
If this function is called when an endpoint is configured in DMA mode 0 the USB controller does not generate an interrupt.

Returns:
None.

8.2.2.21 USBEndpointPacketCountSet

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

Prototype:
```
void
USBEndpointPacketCountSet(uint32_t ui32Base,
                          uint32_t ui32Endpoint,
                          uint32_t ui32Count)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint index to target for this write.
ui32Count is the number of packets to request.

Description:
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.

8.2.2.22 USBEndpointStatus

Returns the current status of an endpoint.

Prototype:
```
uint32_t
USBEndpointStatus(uint32_t ui32Base,
                  uint32_t ui32Endpoint)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.

Description:
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 must 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.

8.2.2.23 USBFIFOAddrGet

Returns the absolute FIFO address for a given endpoint.

Prototype:

```
uint32_t
USBFIFOAddrGet(uint32_t ui32Base,
               uint32_t ui32Endpoint)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint specifies which endpoint's FIFO address to return.

Description:

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.

8.2.2.24 USBFIFOConfigGet

Returns the FIFO configuration for an endpoint.

Prototype:

```
void
USBFIFOConfigGet(uint32_t ui32Base,
                 uint32_t ui32Endpoint,
                 uint32_t *pui32FIFOAddress,
                 uint32_t *pui32FIFOSize,
                 uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

pui32FIFOAddress is the starting address for the FIFO.

pui32FIFOSize is the size of the FIFO as specified by one of the USB_FIFO_SZ_ values.

ui32Flags specifies what information to retrieve from the FIFO configuration.

Description:

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 must not be called for endpoint zero. The *ui32Flags* parameter specifies whether the endpoint's OUT or IN FIFO must be read. If in host mode, the *ui32Flags* parameter must be **USB_EP_HOST_OUT** or **USB_EP_HOST_IN**, and if in device mode, the *ui32Flags* parameter must be either **USB_EP_DEV_OUT** or **USB_EP_DEV_IN**.

Returns:

None.

8.2.2.25 USBFIFOConfigSet

Sets the FIFO configuration for an endpoint.

Prototype:

```
void
USBFIFOConfigSet(uint32_t ui32Base,
                 uint32_t ui32Endpoint,
                 uint32_t ui32FIFOAddress,
                 uint32_t ui32FIFOSize,
                 uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32FIFOAddress is the starting address for the FIFO.

ui32FIFOSize is the size of the FIFO specified by one of the USB_FIFO_SZ_ values.

ui32Flags specifies what information to set in the FIFO configuration.

Description:

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 must not be called for endpoint zero. The *ui32FIFOSize* parameter must be one of the values in the **USB_FIFO_SZ_** values.

The *ui32FIFOAddress* value must 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 starts 64 bytes into the USB controller's FIFO memory. The *ui32Flags* value specifies whether the endpoint's OUT or IN FIFO must 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.

8.2.2.26 USBFIFOFlush

Forces a flush of an endpoint's FIFO.

Prototype:

```
void
USBFIFOFlush(uint32_t ui32Base,
             uint32_t ui32Endpoint,
             uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Flags specifies if the IN or OUT endpoint is accessed.

Description:

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 *ui32Flags* parameter be one of **USB_EP_HOST_OUT**, **USB_EP_HOST_IN**, **USB_EP_DEV_OUT**, or **USB_EP_DEV_IN**.

Returns:

None.

8.2.2.27 USBFrameNumberGet

Get the current frame number.

Prototype:

```
uint32_t
USBFrameNumberGet(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the last frame number received.

Returns:
The last frame number received.

8.2.2.28 USBHostAddrGet

Gets the current functional device address for an endpoint.

Prototype:
```
uint32_t
USBHostAddrGet(uint32_t ui32Base,
               uint32_t ui32Endpoint,
               uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
ui32Flags determines if this is an IN or an OUT endpoint.

Description:
This function returns the current functional address that an endpoint is using to communicate with a device. The *ui32Flags* parameter determines if the IN or OUT endpoint's device address is returned.

Note:
This function must only be called in host mode.

Returns:
Returns the current function address being used by an endpoint.

8.2.2.29 USBHostAddrSet

Sets the functional address for the device that is connected to an endpoint in host mode.

Prototype:
```
void
USBHostAddrSet(uint32_t ui32Base,
               uint32_t ui32Endpoint,
               uint32_t ui32Addr,
               uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
ui32Addr is the functional address for the controller to use for this endpoint.
ui32Flags determines if this is an IN or an OUT endpoint.

Description:
This function configures the functional address for a device that is using this endpoint for communication. This *ui32Addr* parameter is the address of the target device that this endpoint is communicating with. The *ui32Flags* parameter indicates if the IN or OUT endpoint is set.

Note:
This function must only be called in host mode.

Returns:
None.

8.2.2.30 USBHostEndpointConfig

Sets the base configuration for a host endpoint.

Prototype:
```
void
USBHostEndpointConfig(uint32_t ui32Base,
                      uint32_t ui32Endpoint,
                      uint32_t ui32MaxPayload,
                      uint32_t ui32NAKPollInterval,
                      uint32_t ui32TargetEndpoint,
                      uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
ui32MaxPayload is the maximum payload for this endpoint.
ui32NAKPollInterval is the either the NAK timeout limit or the polling interval, depending on the type of endpoint.
ui32TargetEndpoint is the endpoint that the host endpoint is targeting.
ui32Flags are used to configure other endpoint settings.

Description:
This function sets the basic configuration for the transmit or receive portion of an endpoint in host mode. The *ui32Flags* parameter determines some of the configuration while the other parameters provide the rest. The *ui32Flags* 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 *ui32NAKPollInterval* 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 the number of frames between interrupt IN requests to an endpoint and has a range of 1 to 255. For isochronous endpoints this value represents a polling interval of 2 ^ (*ui32NAKPollInterval* - 1) frames. When used as a NAK

timeout, the *ui32NAKPollInterval* value specifies 2 ^ (*ui32NAKPollInterval* - 1) frames before issuing a time out.

There are two special time out values that can be specified when setting the *ui32NAKPollInterval* 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 is 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 *ui32MaxPayload* 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 *ui32MaxPayload* 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 must only be called in host mode.

Returns:
None.

8.2.2.31 USBHostEndpointDataAck

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

Prototype:
```
void
USBHostEndpointDataAck(uint32_t ui32Base,
                       uint32_t ui32Endpoint)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.

Description:
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 must only be called in host mode.

Returns:
None.

8.2.2.32 USBHostEndpointDataToggle

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

Prototype:

```
void
USBHostEndpointDataToggle(uint32_t ui32Base,
                          uint32_t ui32Endpoint,
                          bool bDataToggle,
                          uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint specifies the endpoint to reset the data toggle.

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

ui32Flags specifies whether to set the IN or OUT endpoint.

Description:

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 *ui32Flags* parameter can be **USB_EP_HOST_IN** or **USB_EP_HOST_OUT** to access the desired portion of this endpoint. The *ui32Flags* parameter is ignored for endpoint zero.

Note:

This function must only be called in host mode.

Returns:

None.

8.2.2.33 USBHostEndpointStatusClear

Clears the status bits in this endpoint in host mode.

Prototype:

```
void
USBHostEndpointStatusClear(uint32_t ui32Base,
                           uint32_t ui32Endpoint,
                           uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Endpoint is the endpoint to access.

ui32Flags are the status bits that are cleared.

Description:

This function clears the status of any bits that are passed in the *ui32Flags* parameter. The *ui32Flags* parameter can take the value returned from the USBEndpointStatus() call.

Note:

This function must only be called in host mode.

Returns:
None.

8.2.2.34 USBHostHubAddrGet

Gets the current device hub address for this endpoint.

Prototype:
```
uint32_t
USBHostHubAddrGet(uint32_t ui32Base,
                  uint32_t ui32Endpoint,
                  uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
ui32Flags determines if this is an IN or an OUT endpoint.

Description:
This function returns the current hub address that an endpoint is using to communicate with a device. The *ui32Flags* parameter determines if the device address for the IN or OUT endpoint is returned.

Note:
This function must only be called in host mode.

Returns:
This function returns the current hub address being used by an endpoint.

8.2.2.35 USBHostHubAddrSet

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

Prototype:
```
void
USBHostHubAddrSet(uint32_t ui32Base,
                  uint32_t ui32Endpoint,
                  uint32_t ui32Addr,
                  uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.
ui32Addr is the hub address and port for the device using this endpoint. The hub address must be defined in bits 0 through 6 with the port number in bits 8 through 14.
ui32Flags determines if this is an IN or an OUT endpoint.

Description:
This function configures the hub address for a device that is using this endpoint for communication. The *ui32Flags* 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 must only be called in host mode.

Returns:
None.

8.2.2.36 USBHostMode

Change the mode of the USB controller to host.

Prototype:
```
void
USBHostMode(uint32_t ui32Base)
```

Parameters:
ui32Base specifies the USB module base address.

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

Note:
This function must only be called on microcontrollers that support OTG operation and have the DEVMODOTG bit in the USBGPCS register.

Returns:
None.

8.2.2.37 USBHostPwrConfig

Sets the configuration for USB power fault.

Prototype:
```
void
USBHostPwrConfig(uint32_t ui32Base,
                 uint32_t ui32Flags)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Flags specifies the configuration of the power fault.

Description:
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.

8.2.2.38 USBHostPwrDisable

Disables the external power pin.

Prototype:
```
void
USBHostPwrDisable(uint32_t ui32Base)
```

Parameters:
ui32Base specifies the USB module base address.

Description:
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.

8.2.2.39 USBHostPwrEnable

Enables the external power pin.

Prototype:

```
void
USBHostPwrEnable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

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.

8.2.2.40 USBHostPwrFaultDisable

Disables power fault detection.

Prototype:

```
void
USBHostPwrFaultDisable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function disables power fault detection in the USB controller.

Note:

This function must only be called in host mode.

Returns:

None.

8.2.2.41 USBHostPwrFaultEnable

Enables power fault detection.

Prototype:

```
void
USBHostPwrFaultEnable(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:
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.

8.2.2.42 USBHostRequestIN

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

Prototype:
```
void
USBHostRequestIN(uint32_t ui32Base,
                 uint32_t ui32Endpoint)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.

Description:
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 must only be called in host mode and only for IN endpoints.

Returns:
None.

8.2.2.43 USBHostRequestINClear

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

Prototype:
```
void
USBHostRequestINClear(uint32_t ui32Base,
                      uint32_t ui32Endpoint)
```

Parameters:
ui32Base specifies the USB module base address.
ui32Endpoint is the endpoint to access.

Description:
This function clears a previously scheduled IN transaction if it is still pending. This function is used to safely disable any scheduled IN transactions if the endpoint specified by *ui32Endpoint* is reconfigured for communications with other devices.

Note:
This function must only be called in host mode and only for IN endpoints.

Returns:
None.

8.2.2.44 USBHostRequestStatus

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

Prototype:
```
void
USBHostRequestStatus(uint32_t ui32Base)
```

Parameters:
ui32Base specifies the USB module base address.

Description:
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.

8.2.2.45 USBHostReset

Handles the USB bus reset condition.

Prototype:
```
void
USBHostReset(uint32_t ui32Base,
             bool bStart)
```

Parameters:
ui32Base specifies the USB module base address.
bStart specifies whether to start or stop signaling reset on the USB bus.

Description:
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 must then delay at least 20ms before calling this function again with the *bStart* parameter set to **false**.

Note:
This function must only be called in host mode.

Returns:
None.

8.2.2.46 USBHostResume

Handles the USB bus resume condition.

Prototype:

```
void
USBHostResume(uint32_t ui32Base,
              bool bStart)
```

Parameters:

ui32Base specifies the USB module base address.

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

Description:

When in device mode, this function brings the USB controller out of the suspend state. This call must first be made with the *bStart* parameter set to **true** to start resume signaling. The device application must 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 must first be made with the *bStart* parameter set to **true** to start resume signaling. The host application must 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.

8.2.2.47 USBHostSpeedGet

Returns the current speed of the USB device connected.

Prototype:

```
uint32_t
USBHostSpeedGet(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

This function returns the current speed of the USB bus in host mode.

Example: Get the USB connection speed.

```
//
// Get the connection speed of the device connected to the USB controller.
//
USBHostSpeedGet(USB0_BASE);
```

Note:

This function must only be called in host mode.

Returns:

Returns one of the following: **USB_LOW_SPEED**, **USB_FULL_SPEED**, or **USB_UNDEF_SPEED**.

8.2.2.48 USBHostSuspend

Puts the USB bus in a suspended state.

Prototype:

```
void
USBHostSuspend(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

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

Note:

This function must only be called in host mode.

Returns:

None.

8.2.2.49 USBIntDisableControl

Disables control interrupts on a given USB controller.

Prototype:

```
void
USBIntDisableControl(uint32_t ui32Base,
                     uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Flags specifies which control interrupts to disable.

Description:

This function disables the control interrupts for the USB controller specified by the *ui32Base* parameter. The *ui32Flags* parameter specifies which control interrupts to disable. The flags passed in the *ui32Flags* parameters must be the definitions that start with **USB_INTCTRL_**$*$ and not any other **USB_INT** flags.

Returns:

None.

8.2.2.50 USBIntDisableEndpoint

Disables endpoint interrupts on a given USB controller.

Prototype:

```
void
USBIntDisableEndpoint(uint32_t ui32Base,
                      uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Flags specifies which endpoint interrupts to disable.

Description:

This function disables endpoint interrupts for the USB controller specified by the *ui32Base* parameter. The *ui32Flags* parameter specifies which endpoint interrupts to disable. The flags passed in the *ui32Flags* parameters must be the definitions that start with **USB_INTEP_**$*$ and not any other **USB_INT** flags.

Returns:

None.

8.2.2.51 USBIntEnableControl

Enables control interrupts on a given USB controller.

Prototype:

```
void
USBIntEnableControl(uint32_t ui32Base,
                    uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Flags specifies which control interrupts to enable.

Description:

This function enables the control interrupts for the USB controller specified by the *ui32Base* parameter. The *ui32Flags* parameter specifies which control interrupts to enable. The flags passed in the *ui32Flags* parameters must be the definitions that start with **USB_INTCTRL_**$*$ and not any other **USB_INT** flags.

Returns:

None.

8.2.2.52 USBIntEnableEndpoint

Enables endpoint interrupts on a given USB controller.

Prototype:

```
void
USBIntEnableEndpoint(uint32_t ui32Base,
                     uint32_t ui32Flags)
```

Parameters:

ui32Base specifies the USB module base address.

ui32Flags specifies which endpoint interrupts to enable.

Description:

This function enables endpoint interrupts for the USB controller specified by the *ui32Base* parameter. The *ui32Flags* parameter specifies which endpoint interrupts to enable. The flags

passed in the *ui32Flags* parameters must be the definitions that start with **USB_INTEP_**$*$ and not any other **USB_INT** flags.

Returns:
None.

8.2.2.53 USBIntRegister

Registers an interrupt handler for the USB controller.

Prototype:
```
void
USBIntRegister(uint32_t ui32Base,
               void (*pfnHandler)(void))
```

Parameters:
ui32Base specifies the USB module base address.
pfnHandler is a pointer to the function to be called when a USB interrupt occurs.

Description:
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.

8.2.2.54 USBIntStatus

Returns the control interrupt status on a given USB controller.

Prototype:
```
uint32_t
USBIntStatus(uint32_t ui32Base,
             uint32_t *pui32IntStatusEP)
```

Parameters:
ui32Base specifies the USB module base address.
ui32IntStatusEP is a pointer to the variable which holds the endpoint interrupt status from RXIS And TXIS.

Description:
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 are 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. This is the value of USBIS.

8.2.2.55 USBIntStatusControl

Returns the control interrupt status on a given USB controller.

Prototype:
```
uint32_t
USBIntStatusControl(uint32_t ui32Base)
```

Parameters:
ui32Base specifies the USB module base address.

Description:
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 are 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.

8.2.2.56 USBIntStatusEndpoint

Returns the endpoint interrupt status on a given USB controller.

Prototype:
```
uint32_t
USBIntStatusEndpoint(uint32_t ui32Base)
```

Parameters:
ui32Base specifies the USB module base address.

Description:
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 are 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.

8.2.2.57 USBIntUnregister

Unregisters an interrupt handler for the USB controller.

Prototype:
```
void
USBIntUnregister(uint32_t ui32Base)
```

Parameters:
ui32Base specifies the USB module base address.

Description:
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.

8.2.2.58 USBModeGet

Returns the current operating mode of the controller.

Prototype:
```
uint32_t
USBModeGet(uint32_t ui32Base)
```

Parameters:
ui32Base specifies the USB module base address.

Description:
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**.

8.2.2.59 USBNumEndpointsGet

Returns the number of USB endpoint pairs on the device.

Prototype:

```
uint32_t
USBNumEndpointsGet(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

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.

8.2.2.60 USBOTGMode

Change the mode of the USB controller to OTG.

Prototype:

```
void
USBOTGMode(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

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.

8.2.2.61 USBOTGSessionRequest

Starts or ends a session.

Prototype:

```
void
USBOTGSessionRequest(uint32_t ui32Base,
                     bool bStart)
```

Parameters:

ui32Base specifies the USB module base address.

bStart specifies if this call starts or ends a session.

Description:

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.

8.2.2.62 USBPHYPowerOff

Powers off the USB PHY.

Prototype:

```
void
USBPHYPowerOff(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

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.

8.2.2.63 USBPHYPowerOn

Powers on the USB PHY.

Prototype:

```
void
USBPHYPowerOn(uint32_t ui32Base)
```

Parameters:

ui32Base specifies the USB module base address.

Description:

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.

8.3 Programming Example

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 USBFIFOConfigSet() 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