µEZ™ Software User's Guide

FDI Future Designs, Inc.
Your Development Partner
2702 Triana Boulevard SW, Huntsville, AL 35805

FDI PN: MA00016
Revision: 1.05, 5/25/2010 11:08:00 AM
Printed in the United States of America

Revision History

Who	Date	Details
L. Shields	9/10/2008	Initial version.
L. Shields	10/8/2008	UEZSystemInit() added. UEZTaskDelay() changed to only delay current task.
P. Quirk	10/10/2008	Updated introductory µEZ™ section, renumbered sections to segment OSAL from Subsystem API, recommendation in section 3 to remove UEZ prefix on functions, parameters, and return codes.
L. Shields	10/15/2008	Added diagram. Updated documents.
L. Shields	10/28/2008	Added extensive information about existing subsystem device drivers and HAL drivers.
L. Shields	10/29/2008	Added serial/stream docs. Added HAL and device driver struct definitions for each system. Added example widget drivers and library in appendix.
L. Shields	11/18/2008	Finally replaced all uFlex with µEZ™
L. Shields	11/19/2008	Added file I/O routines (UEZFile…)
L. Shields	11/26/2008	Added lwIP TCP/IP functions.
L. Shields	12/3/2008	Added I2C routines
L. Shields	12/5/2008	Added SWIM library routines.
L. Shields	1/12/2009	Added USB Device HAL driver and USB Device Driver information. SetPalette routines added to LCD drivers. Touch sensitivity routines added to touch screen drivers and library. Added FileStart/Stop/Next routines for doing file enumeration in a given directory.
L. Shields	2/3/2009	Added RTC Device, HAL, and uEZ API. Added Mass Storage device driver. Added SPI new functions both in device and HAL layers. Added Backlight device driver. Updated LCD device driver Configure() function. Added USB Host device driver API.
L. Shields	2/18/2009	Added Accelerometer, Button Bank, EEPROM, and Temperature device driver. Added PWM HAL driver.
L. Shields	3/24/2009	Change UEZTSClose() to properly remove TS queue. Added boot process section. Added platform and processor creation details.
L. Shields	6/8/2009	Added new RTC driver functions Validate and SetClockOutHz. Modified Platform initialization documented (uEZPlatformStartup).
L. Shields	7/22/2009	Updated for uEZ v1.03. Added more details on platform startup. Formatting changes. Modified introduction for more info about FDI.
L. Shields	9/2/2009	Added Serial and Stream Flush commands
L. Shields	2/20/2010	Added UEZADC, UEZCharDisplay, UEZKeypad UEZLEDBank, UEZTemperature, UEZToneGenerator routines. Added UEZFileGetLength()
L. Shields	4/1/2010	Added notes about SPI and SSP data direction of iDataIn and iDataOut.
L. Shields	4/2/2010	Added HAL_DAC. Added HAL_BatteryRAM.
L. Shields	5/24/2010	Added Flash device driver and system functions.

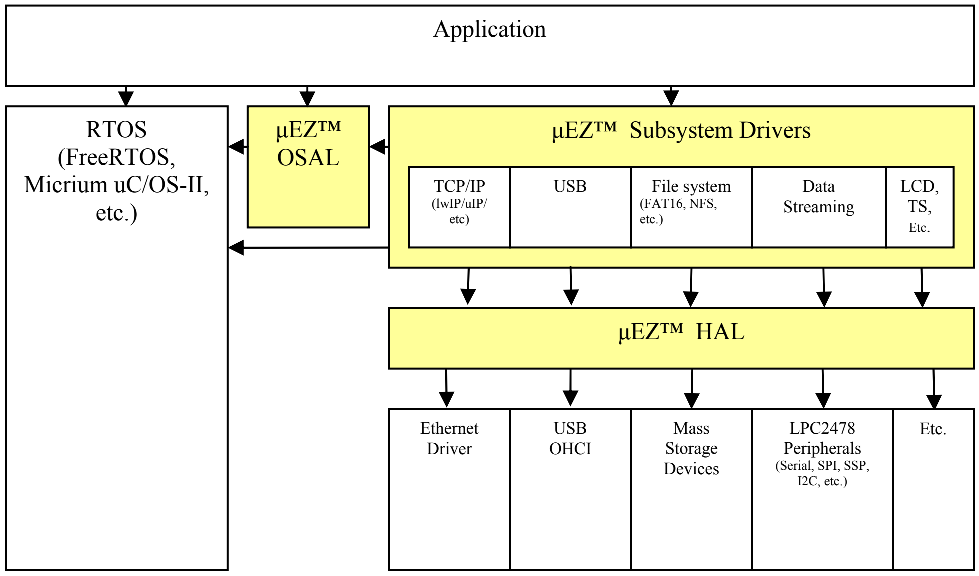
Table of Contents

1.0 µEZ™

µEZ™ is an open source, middleware platform so there is no cost to the user. Customers can directly integrate µEZ™ into their embedded application for free or can contract with Future Designs to provide affordable integration services that are customized to your hardware and software requirements. Future Designs integration services are full turnkey and can cost as little as $10K and may require only 4 weeks of schedule time to complete and test. The goal of the µEZ™ platform is to provide underlying RTOS and processor abstraction enabling the application programmer to focus on the value-added features of their product. µEZ™ is an ***optional*** platform that enhances portability of application code to multiple ARM® platforms with high reusability.

µEZ™ has three primary components which are highlighted in the diagram below:

- Operating System Abstraction Layer (OSAL)
- Subsystem drivers
- Hardware Access Layer (HAL)



The OSAL is the primary component of µEZ™ and provides applications access to the following features in an OS-independent fashion:

- Pre-emptive multitasking core
- Stack overflow detection options
- Unlimited number of tasks
- Queues
- Semaphores (binary, counting, and mutex)

The Subsystem drivers utilize the abstracted RTOS OSAL functions to provide protected access to the processor peripherals. The Subsystem drivers also support direct usage of FreeRTOS when the µEZ™ platform is not utilized. The HAL functions provide single-threaded unprotected access to the processor peripherals. Customers can use the µEZ™ HAL routines provided by Future Designs or can write their own. When written correctly, the HAL routines provide for RTOS and µEZ™ independence.

µEZ™ currently supports the following processor families:

- NXP LPC24xx (LPC2478 is the initial supported processor)
- NXP LPC23xx family
- NXP LPC32xx family
- More processors to come …

µEZ™ currently supports the following RTOS families:

- FreeRTOS (version 5.0.3)
- Micrium uC/OS-II

1.1 Basic Types

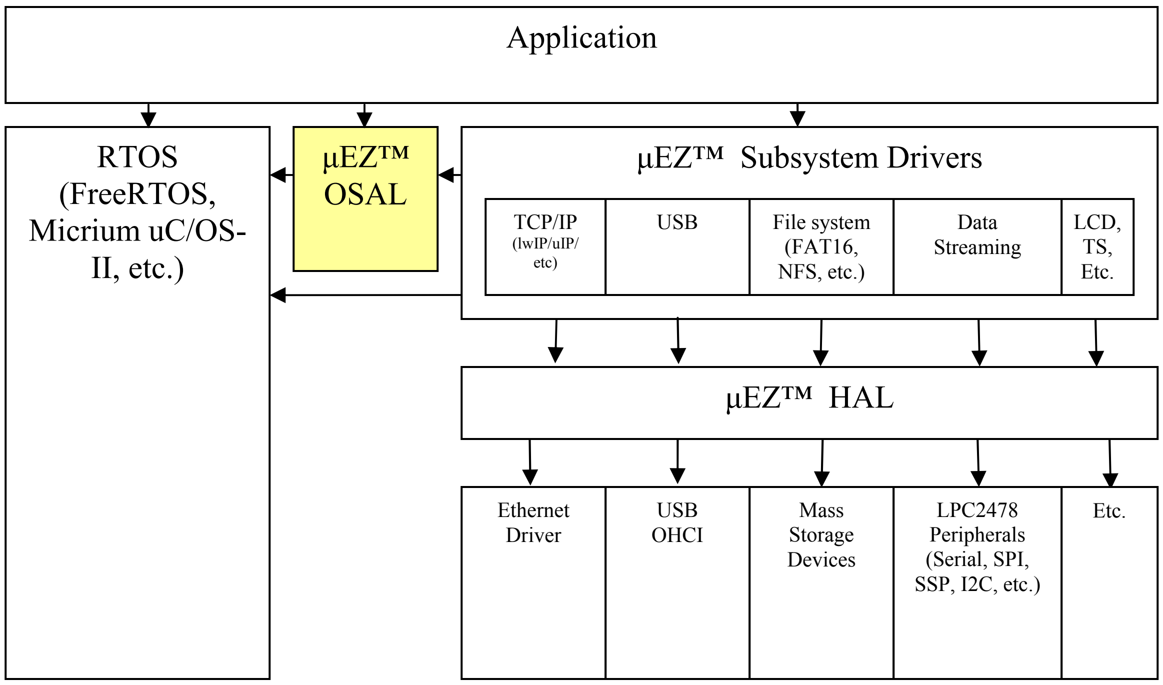
The µEZ™ using the following data types for all of its routines.

Name	Type	Notes
TUInt32	32-bit unsigned integer	
TInt32	32-bit signed integer	
TUInt16	16-bit unsigned integer	
TInt16	16-bit signed integer	
TUInt8	8-bit unsigned integer	
char	8-bit signed integer	
TBool	Boolean value	Use ETrue, or EFalse for values.
T_uezHandle	32-bit handle	Unique identifier for many system wide resources.

2.0 µEZ™ Operating System Access Layer (OSAL)

The µEZ™ condenses all operating system features down to five categories of interfaces:

- Memory
- Tick Counter
- Task
- Queues
- Semaphores



2.1 µEZ™ Memory API

In µEZ™ , all memory comes from a single heap. Standard routines for allocating, freeing, and re-allocating are allowed.

Although the µEZ™ Memory API allows unlimited allocation, resources are limited and memory fragmentation can occur. Applications need to limit memory allocation as much as possible.

2.1.1. UEZMemAlloc()

Description:

Allocate memory from the system heap.

Parameters:

TUInt32 aSize – Number of bytes to allocate

Results:

void * -- Pointer to memory allocated, or NULL if memory not available.

2.1.2. UEZMemFree()

Description:

Free a previously allocated memory block.

Parameters:

void *aMemory – Pointer to memory previously allocated.

Results:

No return values.

2.1.3. UEZMemRealloc()

Description:

Reallocate a block of memory, truncating or copying the contents.

Parameters:

void *aMemory – Pointer to memory previously allocated.
TUInt32 aSize – Number of bytes for new block.

Results:

void * -- Pointer to new memory block. Truncating may still move the block. Returns NULL if the block failed, but the original block will be unmodified.

2.1.4. malloc()

Alias to C compatible routine. See UEZMemAlloc().

2.1.5. free()

Alias to C compatible routine. See UEZMemFree().

2.1.6. realloc()

Alias to C compatible routine. See UEZMemRealloc().

2.2 µEZ™ Tick Counter API

The µEZ™ system provides a 32-bit system counter that runs with a 1 ms interval. This timer is used for timeouts and other time based measurements. The timer starts at 0 at power up and runs constantly never stopping. When it reaches 2^{32}, it rolls over and continues. The accuracy of this timer is the same as the oscillator frequency.

2.2.1. UEZTickCounterGet()

Description:

Returns the tick counter value.

Parameters:

None

Results:

TUInt32 – 32 bit value of counter.

2.2.2. UEZTickCounterGetDelta()

Description:

Returns the amount of time that has occurred (in ticks) since a previous tick reading.

Parameters:

TUInt32 aStart – Previous tick reading.

Results:

TUInt32 – Number of ticks of time that has elapsed since previous tick reading.

2.3 µEZ™ Task API

The µEZ™ system uses a very simple multi-tasking environment. Each task has its own identifier, stack, and main function. When execution of the task function completes, the task is automatically deleted and recycled. Parameters can be passed into the function.

2.3.1. UEZTaskCreate()

Description:

Create a new task of execution and immediately start running.

Parameters:

T_uezTaskFunction aFunction – Function called when executing. If this function exits, the task is automatically deleted.

const TUInt8 * const aName – Unique identifier for this task.

TUInt32 aStackSize – Stack size needed by task. Do not add bytes for overhead. The uEZ system automatically adds additional bytes for overhead.

void *aParameters – Pointer to data that is passed to the T_uezTaskFunction routine. Can be any value including NULL or 0.

T_uezPriority aPriority – Priority level of this task.

T_uezTask *aCreatedTask – Pointer to place the created T_uezTask handle. If the task is created, this value will hold the handle. If the task is not created, the value will be 0.

Results:

T_uezError – On success, returns UEZ_ERROR_NONE. If there are no more task handless left in the system, the routine will return UEZ_ERROR_TOO_MANY_TASKS. If there is no more stack space available, UEZ_ERROR_OUT_OF_STACK_SPACE will be returned.

2.3.2. UEZTaskDelay()

Description:

Delays the current task for a given number of timer ticks.

Parameters:

TUInt32 aTicks – Number of timer ticks to delay.

Results:

T_uezError – On success, returns UEZ_ERROR_NONE. If invalid handle or handle to an already deleted task, returns UEZ_ERROR_HANDLE_INVALID.

2.3.3. UEZTaskDelete()

Description:

Delete a currently running task. Reclaims the stack space. Note, if the task function has exited, the task handle is automatically deleted.

Parameters:

T_uezTask aTask – Handle to the running task.

Results:

T_uezError – On success, returns UEZ_ERROR_NONE. If invalid handle or handle to an already deleted task, returns UEZ_ERROR_HANDLE_INVALID.

2.3.4. (*T_uezTaskFunction)()

Description:

Tasks must start with this type of function. Exiting this function will automatically delete the task.

Parameters:

T_uezTask aTask – Handle to this task.
void *aParameters – Pointer to a block of parameters, or NULL for no parameters.

Results:

None

2.4 µEZ™ Semaphore API

The µEZ™ system uses provides for three types of semaphores:

- Binary
- Counting
- Mutex

All semaphores use a grab/release mechanism that allow blocking with timeouts, blocking indefinitely, or non-blocking.

2.4.1. UEZSemaphoreCreateBinary()

Description:

Creates a semaphore that can only be grabbed once. Multiple attempts to grab this semaphore will block.

Parameters:

T_uezSemaphore *aSemaphore – Pointer to place created semaphore.

Results:

T_uezError – On success, returns UEZ_ERROR_NONE. If the system cannot create the semaphore, returns UEZ_ERROR_OUT_OF_HANDLES.

2.4.2. UEZSemaphoreCreateCounting()

Description:

Creates a counting semaphore that starts with the full count. Grabs can occur until the counting semaphore reaches zero and then blocks. Releases increment the counting back up.

Parameters:

T_uezSemaphore *aSemaphore – Pointer to place created semaphore.
TUInt32 aCount – Initial count of semaphore

Results:

T_uezError – On success, returns UEZ_ERROR_NONE. If the system cannot create the semaphore, returns UEZ_ERROR_OUT_OF_HANDLES.

2.4.3. UEZSemaphoreCreateMutex()

Description:

Creates a mutex semaphore similar to a binary semaphore. Multiple attempts to grab the mutex will block, but priority inversion may occur. If a lower level task has the mutex grabbed than the requesting task, then the lower level task is raised to a higher priority level.

Releases returns the semaphore and returns the task back to its original priority level.

Parameters:

T_uezSemaphore *aSemaphore – Pointer to place created semaphore.

TUInt32 aCount – Initial count of semaphore

Results:

T_uezError – On success, returns UEZ_ERROR_NONE. If the system cannot create the semaphore, returns UEZ_ERROR_OUT_OF_HANDLES.

2.4.4. UEZSemaphoreDelete()

Description:

Deletes a previously created semaphore. Use with care. All tasks using the semaphore should be deleted first.

Parameters:

T_uezSemaphore aSemaphore – Previously created semaphore.

Results:

T_uezError – On success, returns UEZ_ERROR_NONE. If the handle is not a proper semaphore (or it has already been deleted), returns UEZ_ERROR_HANDLE_INVALID.

2.4.5. UEZSemaphoreGrab()

Description:

Grab a semaphore. If the semaphore is not available, block or return immediately (depending on value in aTimeout).

Parameters:

T_uezSemaphore aSemaphore – Semaphore to grab.

TUInt32 aTimeout – Number of ticks to timeout if grab is unsuccessful. If a value of UEZ_TIMEOUT_NONE is used, the grab does not wait, but returns immediately. If the value UEZ_TIMEOUT_INFINITE is used, the grab will only return when the grab is successful.

Results:

T_uezError – On success, returns UEZ_ERROR_NONE. If the handle is not a proper semaphore (or it has already been deleted), returns UEZ_ERROR_HANDLE_INVALID. If the grab times out, returns UEZ_ERROR_TIMEOUT.

2.4.6. UEZSemaphoreRelease()

Description:

Release a previously grabbed semaphore.

Parameters:

T_uezSemaphore aSemaphore – Semaphore to release.

Results:

T_uezError – On success, returns UEZ_ERROR_NONE. If the handle is not a proper semaphore (or it has already been deleted), returns UEZ_ERROR_HANDLE_INVALID. If the semaphore was not first grabbed, returns UEZ_ERROR_ILLEGAL_OPERATION.

2.5 µEZ™ Queue API

The µEZ™ system supports queues for passing same-size data units between tasks. When data is being put into the queue and the queue is full, the caller can choose to block until the queue is free. Likewise, if a queue is empty, a caller can block when getting data out of the queue. Memory for the queue is allocated once at creation time and freed when the queue is deleted.

2.5.1. UEZQueueCreate()

Description:

Create a queue and allocate the memory to be used with it.

Parameters:

TUInt32 aNumItems – Number of items in the queue.
TUInt32 aItemSize – Size (in bytes) of each item in the queue.
T_uezQueue *aQueue – Pointer to place queue handle. The value will be 0 if could not create.

Results:

T_uezError – On success, returns UEZ_ERROR_NONE and fills aQueue with the handle to the new queue. If the system cannot create the queue due to insufficient handles, returns UEZ_ERROR_OUT_OF_HANDLES. If the queue cannot allocate the memory for the queue, it returns UEZ_ERROR_OUT_OF_MEMORY.

2.5.2. UEZQueueDelete()

Description:

Delete a previously created queue and free the queue's memory.

Parameters:

T_uezQueue aQueue – Previously created queue.

Results:

T_uezError – On success, returns UEZ_ERROR_NONE. If a non-queue handle is passed, returns UEZ_ERROR_HANDLE_INVALID.

2.5.3. UEZQueueGetCount()

Description:

Returns the number of items in the queue.

Parameters:

T_uezQueue aQueue – Previously created queue.
TUInt32 *aCount – Pointer to integer to receive count.

Results:

T_uezError – On success (or before the timeout), returns UEZ_ERROR_NONE. If a non-queue handle is passed, returns UEZ_ERROR_HANDLE_INVALID.

2.5.4. UEZQueuePeek()

Description:

Copies the data from the front of the queue or blocks waiting for the space.

Parameters:

T_uezQueue aQueue – Previously created queue.

void *aItem – Pointer to memory to receive copied front item (of size equal to aItemSize as provided in UEZQueueCreate).

TUInt32 aTimeout -- Number of ticks to timeout if receive is unsuccessful. If a value of UEZ_TIMEOUT_NONE is used, the send does not wait, but returns immediately. If the value UEZ_TIMEOUT_INFINITE is used, the receive will wait forever until the space is available.

Results:

T_uezError – On success (or before the timeout), returns UEZ_ERROR_NONE. If a non-queue handle is passed, returns UEZ_ERROR_HANDLE_INVALID. If a timeout occurred, returns UEZ_ERROR_TIMEOUT.

2.5.5. UEZQueueReceive()

Description:

Removes data from the front of the queue or blocks waiting for the space.

Parameters:

T_uezQueue aQueue – Previously created queue.

void *aItem – Pointer to memory to receive memory received (of size equal to aItemSize as provided in UEZQueueCreate).

TUInt32 aTimeout -- Number of ticks to timeout if receive is unsuccessful. If a value of UEZ_TIMEOUT_NONE is used, the send does not wait, but returns immediately. If the value UEZ_TIMEOUT_INFINITE is used, the receive will wait forever until the space is available.

Results:

T_uezError – On success (or before the timeout), returns UEZ_ERROR_NONE. If a non-queue handle is passed, returns UEZ_ERROR_HANDLE_INVALID. If a timeout occurred, returns UEZ_ERROR_TIMEOUT.

2.5.6. UEZQueueSend()

Description:

Adds data to the back end of the queue or blocks waiting for the space.

Parameters:

T_uezQueue aQueue – Previously created queue.

void *aItem – Pointer to data to add to queue (of size equal to aItemSize as provided in UEZQueueCreate).

TUInt32 aTimeout -- Number of ticks to timeout if send is unsuccessful. If a value of UEZ_TIMEOUT_NONE is used, the send does not wait for space, but returns immediately. If the value UEZ_TIMEOUT_INFINITE is used, the send will wait forever until the space is available.

Results:

T_uezError – On success (or before the timeout), returns UEZ_ERROR_NONE. If a non-queue handle is passed, returns UEZ_ERROR_HANDLE_INVALID. If a timeout occurred, returns UEZ_ERROR_TIMEOUT.

2.5.7. UEZQueueSendFront()

Description:

Adds data to the front end (instead of back end) of the queue or blocks waiting for the space.

Parameters:

T_uezQueue aQueue – Previously created queue.

void *aItem – Pointer to data to add to queue (of size equal to aItemSize as provided in UEZQueueCreate).

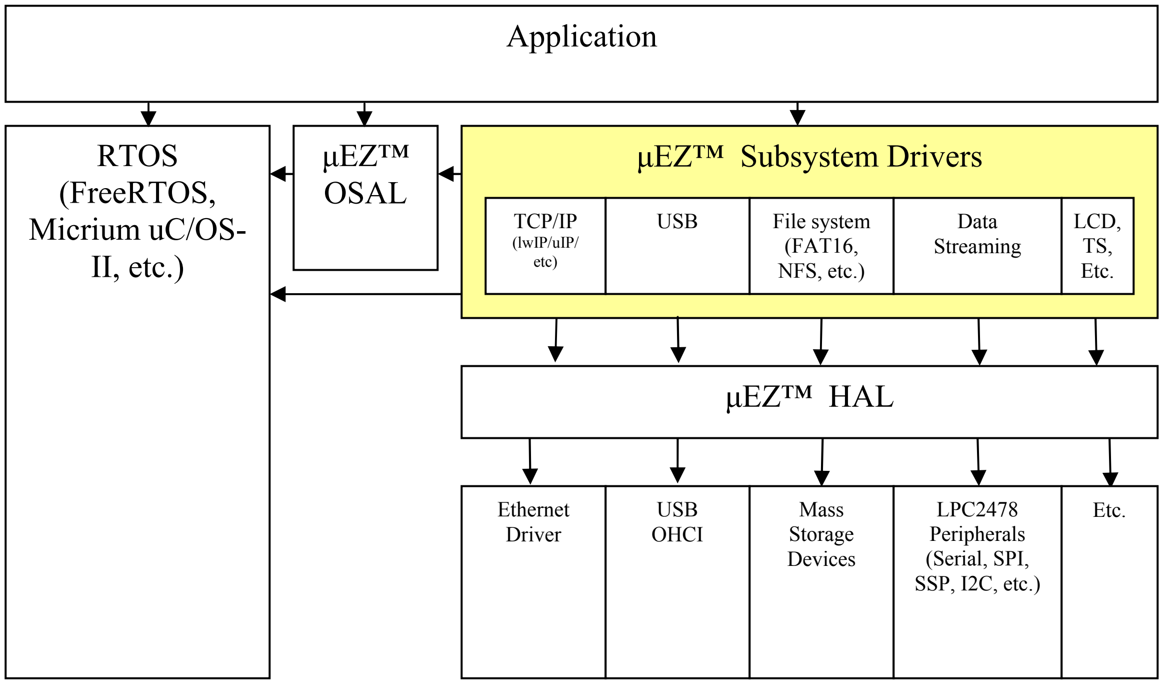
TUInt32 aTimeout -- Number of ticks to timeout if send is unsuccessful. If a value of UEZ_TIMEOUT_NONE is used, the send does not wait for space, but returns immediately. If the value UEZ_TIMEOUT_INFINITE is used, the send will wait forever until the space is available.

Results:

T_uezError – On success (or before the timeout), returns UEZ_ERROR_NONE. If a non-queue handle is passed, returns UEZ_ERROR_HANDLE_INVALID. If a timeout occurred, returns UEZ_ERROR_TIMEOUT.

3.0 Subsystem driver API

The µEZ™ Subsystem drivers wrap around existing functions and provide a common API to commonly used devices. Although not required, the API provides a good mechanism for code re-use.



3.1 µEZ™ ADC API

The µEZ™ system supplies routines to synchronize accesses to do ADC reads of the target system.

3.1.1. UEZADCClose()

Description:

Ends access to an ADC bank.

Parameters:

T_uezDevice aDevice –Handle to an open ADC bank device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.1.2. UEZADCOpen()

Description:

Open access to the ADC bank. Access is not mutually exclusive to other tasks, but is thread protected. A bank is composed of several channels.

Parameters:

const char *const aName – Name of ADC bank to access.

T_uezDevice *aDevice – Returned handle to opened ADC device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.1.3. UEZADCRequestSingle ()

Description:

Perform an ADC channel read action per the given request structure.

Parameters:

T_uezDevice aDevice –Handle to open ADC bank device.

ADC_RequestSingle *aRequest – ADC request type specifying channel, bit size, trigger (usually ADC_TRIGGER_NOW), falling edge (if ADC_TRIGGER_GPIO_x), and a pointer to the data captured. The data returned is in the lower bits of the receiving field.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.2 µEZ™ CharDisplay API

The µEZ™ system supplies routines to synchronize accesses to do a character display. A character display is a grid of characters. The current implementation assumes a left to right orientation and characters are considered fixed width with a limited number of columns and rows. The underlying communication and timing is hidden at this level.

3.2.1. UEZCharDisplayClearRow()

Description:

Clears a single line of characters on the screen by drawing the given character across all the columns of a single row.

Parameters:

T_uezDevice aDevice –Handle to open character display device.
TUInt32 aRow – Row within display to clear
TUInt32 aChar – Character to draw in each space

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID

3.2.2. UEZCharDisplayClearScreen()

Description:

Clears the character display with empty/space characters.

Parameters:

T_uezDevice aDevice –Handle to open character display device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If clear screen command is not supported, returns UEZ_ERROR_NOT_SUPPORTED.

3.2.3. UEZCharDisplayClose()

Description:

Ends access to a character display.

Parameters:

T_uezDevice aDevice –Handle to an open character display device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.2.4. UEZCharDisplayControl()

Description:

Send a device specific display control command to the character display. The exact operation of this command is device specific.

Parameters:

T_uezDevice aDevice –Handle to open character display device.
TUInt32 aControl – Control code
void *aControlData – Pointer to data being sent and/or received.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If no control codes are supported, returns UEZ_ERROR_NOT_SUPPORTED.

3.2.5. UEZCharDisplayDrawChar()

Description:

Draw a single character on the character display at the given column and row. If a column or row is specified outside the range of the device, it is not drawn.

Parameters:

T_uezDevice aDevice –Handle to open character display device.
TUInt32 aRow – the vertical position on the character display (0 is the top row).
TUInt32 aColumn – the horizontal position on the character display (0 is the left).
TUInt32 aChar – The character to be placed in the given column and row.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.2.6. UEZCharDisplayDrawString()

Description:

Draw a string of zero terminated characters from left to right at the given location.

Parameters:

T_uezDevice aDevice –Handle to open character display device.
TUInt32 aRow – the vertical position on the character display (0 is the top row).
TUInt32 aColumn – the horizontal position on the character display (0 is the left).
const char *aChar – A string of 8-bit characters to be drawn left to right.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.2.7. UEZCharDisplayGetSize()

Description:

Gets the size of the display in the number of visible rows and columns are available on the display.

Parameters:

T_uezDevice aDevice –Handle to open character display device.
TUInt32 *aNumRows – Place to store number of rows
TUInt32 *aNumColumns – Place to store number of columns

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.2.8. UEZCharDisplayOpen()

Description:

Open access to the character display. Access is not mutually exclusive to other tasks, but is thread protected.

Parameters:

const char *const aName – Name of character display to access.
T_uezDevice *aDevice – Returned handle to opened character display device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.3 µEZ™ Flash Device Driver API

The µEZ™ system supplies routines to synchronize accesses to flash devices. A flash device is defined as a non-volatile memory grouped into blocks that must be erased before they can be programmed. Routines have been provided to get the size, read, write, and erase the flash device.

3.3.1. UEZFlashBlockErase()

Description:

Erase a range of blocks under the given range of bytes. This routine is aggressive and will erase a complete block if only a single byte touches the block. Therefore, knowledge of the blocks must be used to ensure that no more than expected is erased. See routine GetChipInfo for more details.

Parameters:

T_uezDevice aDevice – Opened flash device
TUInt32 aOffset – Offset into flash device (0 address is start of device)
TUInt32 aNumBytes – Minimum number of bytes to erase

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

3.3.2. UEZFlashChipErase()

Description:

Erase the complete flash device.

Parameters:

T_uezDevice aDevice – Opened flash device

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

3.3.3. UEZFlashClose()

Description:

Close a previously opened flash device.

Parameters:

T_uezDevice aDevice – Opened flash device

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

3.3.4. UEZFlashGetBlockInfo()

Description:

Returns information about the specific block under the given offset in the form of T_FlashBlockInfo

```
typedef struct {
    TUInt32 iOffset;
    TUInt32 iSize;
} T_FlashBlockInfo;
```

The iOffset is the block's starting byte offset and iSize is the size of the block (in bytes).

Parameters:

T_uezDevice aDevice – Opened flash device
TUInt32 aOffset – Byte offset into flash device of block
T_FlashBlockInfo * aBlockInfo

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

3.3.5. UEZFlashGetChipInfo()

Description:

Return information about this Flash device. See T_FlashChipInfo below:

```
typedef struct {
    TUInt32 iNumBytesLow;
    TUInt32 iNumBytesHigh; // if bigger than 4 Gig
    TUInt8 iBitWidth;   // 8, 16, or 32

    TUInt32 iNumRegions;
    T_FlashChipRegion iRegions[FLASH_CHIP_MAX_TRACKED_REGIONS];
} T_FlashChipInfo;

typedef struct {
    TUInt32 iNumEraseBlocks;        // count, base 1
    TUInt32 iSizeEraseBlock;        // in bytes
} T_FlashChipRegion;
```

The T_FlashChipInfo structure contains a 64-bit size in bytes, the number of bits wide, and a number of region structures. A region is a sequential grouping of blocks of the same size. The flash device's offset starts at 0 and erase region occur one after another. For example, if a flash chip has 128 blocks of size 1024 bytes and then 8 blocks of size 256 bytes, the first region is at address 0x0, the second region is at address 0x20000, and the last byte is at 0x207FF.

Parameters:

T_uezDevice aDevice – Opened flash device

T_FlashChipInfo *aInfo – Structure filled with information

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

3.3.6. UEZFlashOpen()

Description:

Opens access to the Flash memory. Once open, the flash memory is expected only to be accessed with flash read and write commands and not accessed directly.

Parameters:

const char *const aName – Name of flash device

T_uezDevice *aDevice – Device handle returned

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

3.3.7. UEZFlashRead()

Description:

Safely read a number of bytes from the flash device to memory. If the flash device is not open, the flash device can be accessed directly.

Parameters:

T_uezDevice aDevice – Opened flash device
TUInt32 aOffset – Offset into Flash with 0 being the base byte address.
TUInt8 *aBuffer – Place to store bytes read
TUInt32 aNumBytes – Number of bytes to read

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

3.3.8. UEZFlashQueryReg()

Description:

Read back the result of an internal register of the flash device (if available).

Parameters:

T_uezDevice aDevice – Opened flash device
TUInt32 aReg – Register index to read
TUInt32 *aValue – Place to return result

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

3.3.9. UEZFlashWrite()

Description:

Safely write a number of bytes from memory into the flash device.
NOTE: The flash memory should first be erased before writing. Reports an error if the memory cannot be written.

Parameters:

T_uezDevice aDevice – Opened flash device
TUInt32 aOffset – Offset into Flash with 0 being the base byte address.
TUInt8 *aBuffer – Bytes to write
TUInt32 aNumBytes – Number of bytes to write

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

3.4 µEZ™ File API

The µEZ™ system supplies a simple file based system for handling data. The µEZ™ File system uses a mount based system with capability to support multiple file system types. Functionality is aimed at simple read/write access for applications that need configuration or logging features.

File systems can be mounted dynamically in the file hierarchy.

NOTE: In version µEZ™ 0.04, currently only one file system can be mounted and directories are not used. The mount path is currently not used. All files should be specified without directory paths.

3.4.1. UEZFileClose()

Description:

Closes a previously opened file.

Parameters:

T_uezFile aFile – File that was previously opened.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the file handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.4.2. UEZFileDelete()

Description:

Deletes a file. File name must have the complete path to the file.

Parameters:

const char * const aName – File name of file to delete

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the file handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If the file cannot be found, returns UEZ_ERROR_NOT_AVAILABLE.

3.4.3. UEZFileFindStart()

Description:

Start looking for files in the given directory.

Parameters:

const char * const aDirectory – name of directory to look within.

T_uezFileEntry *aEntry – Current file entry.

Results:

T_uezError -- UEZ_ERROR_NOT_AVAILABLE if the drive is not mounted. UEZ_ERROR_NONE if the first file is found. UEZ_ERROR_NOT_FOUND if no files found.

3.4.4. UEZFileFindNext()

Description:

Seek the next file in the list, or return with an error code.

Parameters:

T_uezFileEntry *aEntry – Current file entry.

Results:

T_uezError -- UEZ_ERROR_NOT_AVAILABLE if the drive is not mounted. UEZ_ERROR_NONE if the next file is found. UEZ_ERROR_NOT_FOUND if no more files found.

3.4.5. UEZFileFindStop()

Description:

Stop looking for files and release related memory.

Parameters:

T_uezFileEntry *aEntry – Last file entry.

Results:

T_uezError -- UEZ_ERROR_NOT_AVAILABLE if the drive is not mounted. UEZ_ERROR_NONE if the next file is found. UEZ_ERROR_NOT_FOUND if no more files found.

3.4.6. UEZFileGetLength()

Description:

Return the length of the open file.

Parameters:

T_uezFile aFile – File that was previously opened.

TUInt32 *aLength – Place to store returned length.

Results:

T_uezError -- UEZ_ERROR_NOT_AVAILABLE if the drive is not mounted. UEZ_ERROR_NONE if the next file is found. UEZ_ERROR_NOT_FOUND if no more files found.

3.4.7. UEZFileOpen

Description:

Open a file for reading or writing. File names must be a complete path to the file.

Parameters:

const char * const aName – Name of file to open.

TUInt32 aFlags – FILE_FLAG_READ_ONLY reads a file starting at the beginning. FILE_FLAG_APPEND opens a file for writing at the end of the file. FILE_FLAG_WRITE opens a file for writing, truncating the previous file.

T_uezFile *aFile – Place to store returned file handle.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the file handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If the file cannot be found or opened, returns UEZ_ERROR_NOT_AVAILABLE.

If too many files are opened, returns
UEZ_ERROR_OUT_OF_MEMORY.

3.4.8. UEZFileRead()

Description:

Read a number of (binary) bytes of the file into a given buffer.

Parameters:

T_uezFile aFile – File that was previously opened.
void *aBuffer – Pointer to memory to receive bytes.
TUInt32 aNumBytes – Number of bytes to read.
TUInt32 *aNumBytesRead – Number of bytes actually read.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the file handle is bad, returns UEZ_ERROR_HANDLE_INVALID. Other errors are possible, but depend on the file system underneath.

3.4.9. UEZFileSystemMount()

Description:

Mounts a file system into the file hierarchy.

Parameters:

T_uezDevice aFileSystemDevice – File system device driver to mount.
const char * const aMountPoint – File hierarchy location to mount at.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Returns error code (e.g. UEZ_ERROR_OUT_OF_MEMORY) if cannot mount.

3.4.10. UEZFileSystemUnmount()

Description:

Unmounts a file system from the file hierarchy.

Parameters:

const char * const aMountPoint – Location in file hierarchy where mounted.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Returns UEZ_ERROR_NOT_FOUND if no file system at that location.

3.4.11. UEZFileWrite()

Description:

Write a number of (binary) bytes into the file from the given buffer.

Parameters:

T_uezFile aFile – File that was previously opened.
void *aBuffer – Pointer to memory bytes.
TUInt32 aNumBytes – Number of bytes to write.
TUInt32 *aNumBytesWritten – Number of bytes actually written.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the file handle is bad, returns UEZ_ERROR_HANDLE_INVALID. Other errors are possible, but depend on the file system underneath.

3.5 µEZ™ I2C API

The µEZ™ system supplies routines to synchronize accesses to the I2C busses of the target system.

3.5.1. UEZI2CClose()

Description:

Ends access to an I2C bus.

Parameters:

T_uezDevice aDevice –Handle to an open I2C device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.5.2. UEZI2COpen()

Description:

Open access to the I2C bus. Access is not mutually exclusive to other tasks, but is thread protected.

Parameters:

const char *const aName – Name of I2C bus to access.

T_uezDevice *aDevice – Returned handle to opened I2C device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.5.3. UEZI2CRead()

Description:

Perform an I2C read action and return data bytes to the given buffer.

Parameters:

T_uezDevice aDevice –Handle to open I2C device.

TUInt8 aAddress – 7-bit I2C address (most significant bit is always 0).

TUInt32 aSpeed – Speed of I2C access in kHz. Typically 100 or 400 kHz.

TUInt8 *aData – Place to store read bytes.

TUInt8 aDataLength – Number of bytes to read from device.

TUInt32 aTimeout – General safety timeout to perform action.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If a timeout occurs, returns UEZ_ERROR_TIMEOUT. If the device is not available or stops mid-read, returns UEZ_ERROR_NAK.

3.5.4. UEZI2CTransaction()

Description:

Performs an I2C write and/or read action using the given request structure. This routine can be used when a read action needs to occur immediately after doing a write action and to avoid another task getting access to the bus.

Parameters:

T_uezDevice aDevice –Handle to open I2C device.
I2C_Request *aRequest – Request structure (holding read and write parameters).

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If a timeout occurs, returns UEZ_ERROR_TIMEOUT. If the device is not available or stops mid-write, returns UEZ_ERROR_NAK.

3.5.5. UEZI2CWrite()

Description:

Perform an I2C write action and sends data bytes to the given device.

Parameters:

T_uezDevice aDevice –Handle to open I2C device.
TUInt8 aAddress – 7-bit I2C address (most significant bit is always 0).
TUInt32 aSpeed – Speed of I2C access in kHz. Typically 100 or 400 kHz.
TUInt8 *aData – Place hold bytes to write.
TUInt8 aDataLength – Number of bytes to write to device.
TUInt32 aTimeout – General safety timeout to perform action.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If a timeout occurs, returns UEZ_ERROR_TIMEOUT. If the device is not available or stops mid-write, returns UEZ_ERROR_NAK.

3.6 µEZ™ Keypad API

The µEZ™ system supplies routines to synchronize accesses to a keypad device. Polling the keypad will generate an input event.

3.6.1. UEZKeypadClose()

Description:

Ends access to a keypad.

Parameters:

T_uezDevice aDevice –Handle to an open keypad device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.6.2. UEZKeypadOpen()

Description:

Open access to the keypad. Access is not mutually exclusive to other tasks, but is thread protected.

Parameters:

const char *const aName – Name of keypad to access.

T_uezDevice *aDevice – Returned handle to opened keypad device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.6.3. UEZKeypadRead()

Description:

Reads the next waiting keypad input event or waits for another one to occur.

Parameters:

T_uezDevice aDevice –Handle to open keypad device.

T_uezInputEvent *aEvent – Input event received, if any.

TUInt32 aTimeout – Timeout in milliseconds or UEZ_TIMEOUT_NONE.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE and fills aEvent. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If a timeout occurs, returns UEZ_ERROR_TIMEOUT.

3.7 µEZ™ LCD API

The µEZ™ system supplies a direct access memory model for LCDs. The LCD memory is arranged as a linear array of pixels. Each pixel can take up any number of bits or bytes depending on the model of the LCD display. If available, the LCD also offers multiple frames of data.

3.7.1. UEZLCDBacklight()

Description:

Turns on or off the backlight. A value of 0 is off and values of 1 or higher is higher levels of brightness (dependent on the LCD display). If a display is on/off only, this value will be 1 or 0 respectively.

Parameters:

T_uezDevice aDevice – Handle to open LCD device.
TUInt32 aLevel – Backlight intensity.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If the backlight intensity level is invalid, returns UEZ_ERROR_OUT_OF_RANGE.

3.7.2. UEZLCDClose()

Description:

Close previously opened LCD device.

Parameters:

T_uezDevice aDevice – Handle to open LCD device.

Results:

T_uezError – If the device is successfully closed, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.7.3. UEZLCDGetFrame()

Description:

Returns a pointer to the frame memory in the LCD display.

Parameters:

T_uezDevice aDevice – Handle to open LCD device.
TUInt32 aFrame – Index to frame.
void **aFrameBuffer – Pointer to pointer to linear array of memory bytes of memory.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If an invalid device handle, returns UEZ_ERROR_HANDLE_INVALID. If the frame is incorrect, returns UEZ_ERROR_OUT_OF_RANGE.

3.7.4. UEZLCDGetInfo()

Description:

Get information about the LCD device.

Parameters:

T_uezDevice aDevice – Handle to open LCD device.
T_uezLCDConfiguration *aConfiguration – Pointer to store configuration information.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device cannot be found, returns UEZ_ERROR_DEVICE_NOT_FOUND.

3.7.5. UEZLCDOff()

Description:

Turns off the LCD display.

Parameters:

T_uezDevice aDevice – Handle to open LCD device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If an invalid device handle, returns UEZ_ERROR_HANDLE_INVALID.

3.7.6. UEZLCDOn()

Description:

Turns on the LCD display.

Parameters:

T_uezDevice aDevice – Handle to open LCD device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If an invalid device handle, returns UEZ_ERROR_HANDLE_INVALID.

3.7.7. UEZLCDOpen()

Description:

Open and retrieve access the LCD display.

Parameters:

const char * const aName – Unique name of display. Usually "LCD".

T_uezDevice *aDevice – Handle to opened device (or filled with 0 if not opened).

Results:

T_uezError – If the device is opened, returns UEZ_ERROR_NONE. If the device cannot be found, returns UEZ_ERROR_DEVICE_NOT_FOUND.

3.7.8. UEZLCDSetPaletteColor()

Description:

Change the color of a palette entry given the red, green, blue component of a particular color index. The color components are expressed in full 16-bit values at a higher resolution than the hardware can usually perform. The color is down shifted to what the hardware can handle.

Parameters:

T_uezDevice aDevice – Handle to open LCD device.

TUInt32 aColorIndex – Index in the palette.

TUInt16 aRed – Red component (0 – 0xFFFF)

TUInt16 aGreen – Green component (0 – 0xFFFF)

TUInt16 aBlue – Blue component (0 – 0xFFFF)

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If an invalid device handle, returns UEZ_ERROR_HANDLE_INVALID. If the color index is incorrect, returns UEZ_ERROR_OUT_OF_RANGE.

3.7.9. UEZLCDShowFrame()

Description:

Makes the passed frame the actively viewed frame on the LCD (if not already).

Parameters:

T_uezDevice aDevice – Handle to open LCD device.

TUInt32 aFrame – Index to frame.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If an invalid device handle, returns UEZ_ERROR_HANDLE_INVALID. If the frame is incorrect, returns UEZ_ERROR_OUT_OF_RANGE.

3.8 µEZ™ LED Bank API

The µEZ™ system supplies routines to synchronize accesses to bank of LEDs. If the device has blinking capabilities, this is also supported. Blinking is controlled by blink registers so that LEDs can be synchronized to one specific blink register instead of individually setting the timing of each LED blinking rates.

3.8.1. UEZLEDBankClose()

Description:

Ends access to a LED bank.

Parameters:

T_uezDevice aDevice –Handle to an open keypad device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.8.2. UEZLEDBankBlink()

Description:

Blink a group of LEDs in the LED bank using the given blink register.

Parameters:

T_uezDevice aDevice –Handle to open LED bank device.
TUInt32 aBits – Bit representation of the LEDs to blink (up to 32 LEDs).
TUInt32 aBlinkReg – Blink register to determine blink rate and timing.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE and fills aEvent. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.8.3. UEZLEDBankOn()

Description:

Turn on a group of LEDs in the LED bank.

Parameters:

T_uezDevice aDevice –Handle to open LED bank device.
TUInt32 aBits – Bit representation of the LEDs to turn on (up to 32 LEDs).

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE and fills aEvent. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.8.4. UEZLEDBankOff()

Description:

Turn off a group of LEDs in the LED bank.

Parameters:

T_uezDevice aDevice –Handle to open LED bank device.
TUInt32 aBits – Bit representation of the LEDs to turn off (up to 32 LEDs).

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE and fills aEvent. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.8.5. UEZLEDBankOpen()

Description:

Open access to a LED bank. Access is not mutually exclusive to other tasks, but is thread protected.

Parameters:

const char *const aName – Name of LED bank to access.
T_uezDevice *aDevice – Returned handle to opened LED bank device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.8.6. UEZLEDBlink()

Description:

Command a single LED in the LED bank to blink using one of the blink registers.

Parameters:

T_uezDevice aDevice –Handle to open LED bank device.
TUInt8 aIndex – 0-255 index of LED in bank.
TUInt32 aBlinkReg – Blink register to use.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE and fills aEvent. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.8.7. UEZLEDOn()

Description:

Turn on a single LED in the LED bank.

Parameters:

T_uezDevice aDevice –Handle to open LED bank device.
TUInt8 aIndex – 0-255 index of LED in bank.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE and fills aEvent. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.8.8. UEZLEDOff()

Description:

Turn off a single LED in the LED bank.

Parameters:

T_uezDevice aDevice –Handle to open LED bank device.
TUInt8 aIndex – 0-255 index of LED in bank.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE and fills aEvent. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.8.9. UEZLEDBankSetBlinkRegister()

Description:

Set the timing of one of the blink registers (if available).

Parameters:

T_uezDevice aDevice –Handle to open LED bank device.
TUInt32 aBlinkReg – Blink register to set (0 based)
TUInt32 aPeriod – Blink period in milliseconds
TUInt8 aDutyCycle – On duty cycle 0 (0%) to 255 (100%)

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE and fills aEvent. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.9 µEZ™ Stream API

The µEZ™ system supplies access to streamed data such as serial ports, ethernet ports, or other future devices through the UEZStream routines. Each stream has a unique name. For example, the first serial port is "Serial 0". Once open, the port can be written to and read as needed. Timeouts are used to limit the amount of time a stream blocks waiting for data to be read or written. Port settings (such as baud rate and handshaking) are set using the UEZStreamControl command.

3.9.1. UEZStreamClose()

Description:

Closes previously opened stream.

Parameters:

T_uezDevice aDevice – Stream device handle

Results:

T_uezError – If the device is closed, returns UEZ_ERROR_NONE. If the handle is bad, returns UEZ_ERROR_INVALID_HANDLE.

3.9.2. UEZStreamControl()

Description:

Sends a control request to the stream. The exact controls used depend on the underlying device and are specific to that device's control routine.

Parameters:

T_uezDevice aDevice – Stream device handle

TUInt32 aControl – Control request type to send to stream.

void *aControlData – Pointer to data to use with control (sending or receiving).

Results:

T_uezError – If the device is opened, returns UEZ_ERROR_NONE. If the device cannot be found, returns UEZ_ERROR_NOT_FOUND. If the system is out of handles, it returns UEZ_ERROR_OUT_OF_HANDLES. If the stream is already open by other exclusively, returns UEZ_ERROR_NOT_AVAILABLE.

3.9.3. UEZStreamFlush()

Description:

Flush data written to stream device waiting until its transmit buffer is empty.

Parameters:

T_uezDevice aDevice – Stream device handle

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If handle is invalid, returns UEZ_ERROR_HANDLE_INVALID.

3.9.4. UEZStreamOpen()

Description:

Attempts to open the stream for input/output access.

Parameters:

const char * const aName – Name of stream device to open.
T_uezDevice *aDevice – Pointer to opened stream device handle.

Results:

T_uezError – If the device is opened, returns UEZ_ERROR_NONE. If the device cannot be found, returns UEZ_ERROR_NOT_FOUND. If the system is out of handles, it returns UEZ_ERROR_OUT_OF_HANDLES. If the stream is already open by other exclusively, returns UEZ_ERROR_NOT_AVAILABLE.

3.9.5. UEZStreamRead()

Description:

Read data in from the stream device or timeout.

Parameters:

T_uezDevice aDevice – Stream device handle
void *aData – Pointer to place to store incoming data.
TUInt32 aNumBytes – Number of data bytes to read.
TUInt32 *aNumBytesRead – Number of data bytes actually read.
TUInt32 aTimeout – Amount of time to between data byte received until timeout. If the value is UEZ_TIMEOUT_INFINITE, the stream blocks until all data is received. If the value is zero, the input stream's buffer is emptied up to the requested amount and immediately returns.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If a timeout occurs, returns UEZ_ERROR_TIMEOUT. If handle is invalid, returns UEZ_ERROR_HANDLE_INVALID.

3.9.6. UEZStreamWrite()

Description:

Write data out to the stream device or timeout.

Parameters:

T_uezDevice aDevice – Stream device handle
void *aData – Pointer to data to send.
TUInt32 aNumBytes – Number of data bytes to send.
TUInt32 *aNumBytesWritten – Number of data bytes actually written.
TUInt32 aTimeout – Amount of time to between data byte sends until timeout. If the value is UEZ_TIMEOUT_INFINITE, the stream blocks until all data is sent. If the value is zero, the output stream's buffer is filled and immediately returns.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If a timeout occurs, returns UEZ_ERROR_TIMEOUT. If handle is invalid, returns UEZ_ERROR_HANDLE_INVALID.

3.10 µEZ™ Temperature API

The µEZ™ system supplies routines to synchronize accesses to a temperature device. Reading the temperature polls the hardware and returns a final result.

3.10.1. UEZTemperatureClose()

Description:

Ends access to a temperature device.

Parameters:

T_uezDevice aDevice –Handle to an open temperature device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.10.2. UEZTemperatureOpen()

Description:

Open access to the temperature device. Access is not mutually exclusive to other tasks, but is thread protected.

Parameters:

const char *const aName – Name of temperature device to access.

T_uezDevice *aDevice – Returned handle to opened temperature device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.10.3. UEZTemperatureRead()

Description:

Polls the hardware for the current temperature.

Parameters:

T_uezDevice aDevice –Handle to open temperature device.

TInt32 *aTemperature – Returned temperature value in signed 16.16 format (integer in top signed 16 bits and fraction in lower 16 bits). Temperature returned is usually Celsius unless otherwise noted.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE and fills aEvent. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.11 µEZ™ Tone Generator API

The µEZ™ system supplies routines to synchronize accesses to a tone generator device. These routines are intended to provide the simplest procedural method of playing a tone.

3.11.1. UEZToneGeneratorClose()

Description:

Ends access to a tone generator device.

Parameters:

T_uezDevice aDevice –Handle to an open tone generator device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.11.2. UEZToneGeneratorOpen()

Description:

Open access to the tone generator device. Access is not mutually exclusive to other tasks, but is thread protected.

Parameters:

const char *const aName – Name of tone generator device to access.

T_uezDevice *aDevice – Returned handle to opened tone generator device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.11.3. UEZToneGeneratorPlayTone()

Description:

Plays a tone of the given period (not frequency) and of the given duration in milliseconds. Use the macro TONE_GENERATOR_HZ() to convert frequency values into period values. This routine will not return until the tone has been played over the given duration.

Parameters:

T_uezDevice aDevice –Handle to open temperature device.

TUInt32 aTonePeriod – The period of the tone. Use TONE_GENERATOR_HZ() to convert frequency to period values.

TUInt32 aDuration – Number of milliseconds to play.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE and fills aEvent. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.12 µEZ™ Time Date API

The µEZ™ system supplies routines for handling the time and date.

3.12.1. UEZTimeDateGet (T_uezTimeDate *aTimeDate)

Description:

Get the current time and date from the system clock.

Parameters:

T_uezTimeDate *aTimeDate – Pointer to structure with date and time.

Results:

T_uezError -- If time/date structure retrieved, UEZ_ERROR_NONE, else error code.

3.12.2. UEZTimeDateSet (const T_uezTimeDate *aTimeDate)

Description:

Sets the time and date on the system clock.

Parameters:

T_uezTimeDate *aTimeDate – Pointer to structure with date and time.

Results:

T_uezError -- If time/date structure changed, UEZ_ERROR_NONE, else error code.

3.13 µEZ™ Touch Screen (TS) API

The µEZ™ system supplies a touch screen task that monitors the touch screen display. Either a event queue base system can be used, or a raw request interface. In addition, Calibration routines are provided to help with the screen accuracy.

3.13.1. UEZTSClose()

Description:

Closes the touch screen access.

Parameters:

T_uezDevice aDevice – Previously opened touch screen device.

T_uezQueue aQueue – Queue being used for touch screen events or 0.

Results:

T_uezError – If the device is closed, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.13.2. UEZTSGetReading()

Description:

Forces the touch screen to poll for an immediate reading.

Parameters:

T_uezDevice aDevice – Previously opened touch screen device.

T_uezTSReading *aReading – Pointer to structure of the reading (x, y, pressure, etc.).

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. Returns UEZ_ERROR_TIMEOUT if the hardware is unresponsive.

3.13.3. UEZTSOpen()

Description:

Opens access to the touch screen and optionally registers a queue to receive events.

Parameters:

const char * const aName – Unique name of touch screen device. Usually “Touchscreen”.

T_uezDevice *aDevice – Pointer to device handle to be filled when opened (will be set to 0 if not opened).

T_uezQueue *aEventQueue – Pointer to queue to receive touch screen events or 0 for no queue.

Results:

T_uezError – If the device is opened, returns UEZ_ERROR_NONE. If the device cannot be found, returns UEZ_ERROR_DEVICE_NOT_FOUND. If the queue handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.13.4. UEZTSCalibrationStart()

Description:

Puts the touch screen in calibration mode.

Parameters:

T_uezDevice aDevice – Previously opened touch screen device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID.

3.13.5. UEZTSCalibrationAddReading()

Description:

Adds a data point to the touch screen calibration system.

Parameters:

T_uezDevice aDevice – Previously opened touch screen device (in calibration mode).

T_uezTSReading *aReadingTaken – Reading taken using UEZTSGetReading.

T_uezTSReading *aReadingExpected – Reading that was expected (ideal).

Results:

T_uezError – If done (last calibration point), returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If more data is needed, returns UEZ_ERROR_NOT_ENOUGH_DATA.

3.13.6. UEZTSCalibrationEnd()

Description:

Finishes the calibration and uses it.

Parameters:

T_uezDevice aDevice – Previously opened touch screen device (in calibration mode).

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If the calibration was terminated without enough data points, returns UEZ_ERROR_NOT_ENOUGH_DATA.

3.13.7. UEZTSAddQueue()

Description:

Add queue to receive touch screen events for associate device.

Parameters:

T_uezDevice aDevice – Previously opened touch screen device.

T_uezQueue aEventQueue – Queue to receive events

Results:

T_uezError – If the queue added, returns UEZ_ERROR_NONE. If the queue cannot be added, returns UEZ_ERROR_OUT_OF_MEMORY.

3.13.8. UEZTSRemoveQueue()

Description:

Remove queue receiving touch screen events for associate device.

Parameters:

T_uezDevice aDevice – Previously opened touch screen device.
T_uezQueue aEventQueue – Queue currently receiving events

Results:

T_uezError – If the queue added, returns UEZ_ERROR_NONE. If the queue cannot be added, returns UEZ_ERROR_OUT_OF_MEMORY.

3.13.9. UEZTSSetTouchDetectSensitivity()

Description:

Change the low (press) and high (release) points of the touch detection code. These points are hardware specific, but the values passed are 0-0xFFFF as a scale between 0 V and reference Voltage. Notice that putting a gap between high and low provides a range where the press/release is reported as the previous press/release until the threshold is met.

Parameters:

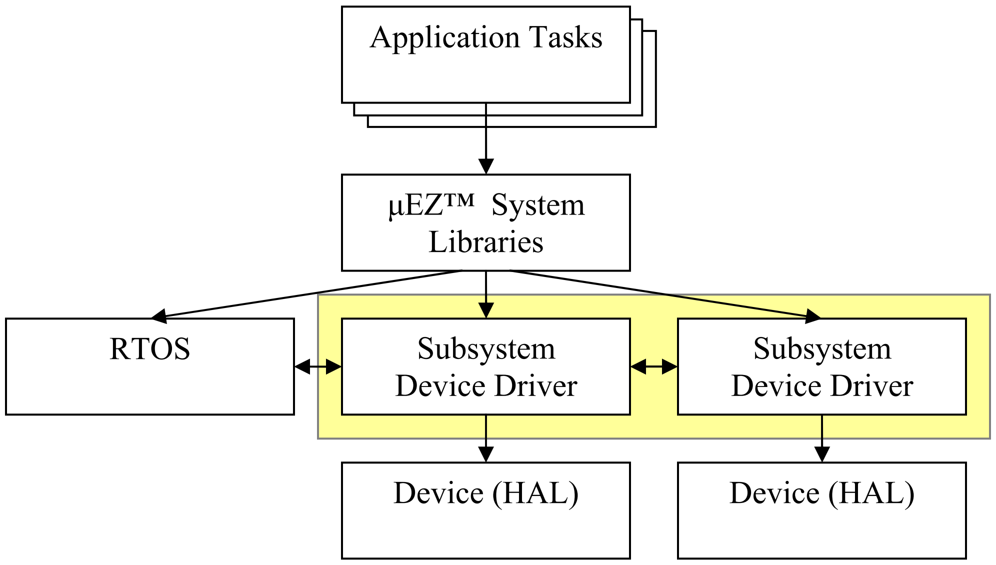
T_uezDevice aDevice – Previously opened touch screen device.
TUInt16 aLowLevel – Low (press) reference level (0-0xFFFF)
TUInt16 aHighLevel – High (press) reference level (0-0xFFFF)

Results:

If successful, return UEZ_ERROR_NONE, else returns error code.

4.0 Subsystem Device Drivers

Subsystem Device Drivers are used to connect low level HAL devices with the RTOS. A unique API is specified for each type of device driver typically closely mapping to underlying HAL device APIs. Subsystem Device Drivers control how the HAL devices interact with the RTOS and application.



Families of Subsystem Device Driver interface types are defined in the directory "Include/Device". For example, a project may be using a serial UART on the processor and a serial UART on the base board. Both of these are stream devices and their subsystem device drivers will use the DEVICE_STREAM interface (see "Include/Device/Stream.h").

Each Subsystem Device Driver must provide a device interface. The device interface is a structure registered at power on and starts with a T_uezDeviceInterface substructure. T_uezDeviceInterface is as follows:

```
typedef struct {
    const char *iName;
    TUInt16 iVersion;   // Version is 0x103 for version 1.03
    T_uezError (*iInitializeWorkspace)(void *aWorkspace);
    TUInt32 iWorkspaceSize;
} T_halInterface;
```

Each field starts with a pointer to a unique name. Names are given in the following format "<device driver type>:<company>:<hardware instance>". For example "SPI:NXP:LPC2478 SPI0" means it is a SPI bus using NXP's LPC2478 processor providing SPI0 support.

Next comes the 16-bit version number. The upper byte is the major version and provides compatibility information – the number must match to be compatible. The lower byte is the minor version. The minor version must be equal or greater to the required minor version to be compatible. For example, version 0x0203 is needed for a project. If the device driver is 0x0300 or 0x0100, the major version is different and definitely not compatible. A device driver of version 0x0200 is also not compatible because it has a lower minor version. However, a device driver of version 0x0204 is compatible even though it may have a few other extra features.

The fields iInitializeWorkspace and iWorkspaceSize are used when the device driver is registered. A workspace is allocated on the heap and calls iInitializeWorkspace to setup the device driver.

The rest of the device interface structure is typically a list of function pointers that expose the features of the device (although other fields can be provided as required by the standard interface).

4.1 Accelerometer Device Driver API (DEVICE_Accelerometer)

The Accelerometer device provides feedback on the forces placed on the target in the X, Y, or Z direction. Orientation of the board can be determined based on these readings in reference to gravity (down).

4.1.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*GetInfo)(
            void *aWorkspace,
            AccelerometerInfo *aInfo);
    T_uezError (*ReadXYZ)(
            void *aWorkspace,
            AccelerometerReading *aReading,
            TUInt32 aTimeout);
} DEVICE_Accelerometer
```

4.1.2. GetInfo()

Description:

Returns a structure containing information about the device. Currently returns the chip ID and the revision number (device specific).

Parameters:

void *aWorkspace – Accelerometer device driver's workspace
AccelerometerInfo *aInfo – Place to store device information

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.1.3. ReadXYZ()

Description:

Reads the X, Y, and Z forces on the unit.

Parameters:

void *aWorkspace – Accelerometer device driver's workspace
AccelerometerReading *aInfo – Place to store reading
TUInt32 aTimeout – Period to keep reading until get a valid result

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Returns UEZ_ERROR_TIMEOUT if could not get a reading in the proper time. Returns an error code in all other cases.

4.2 ADC Bank Device Driver API (DEVICE_ADCBank)

The ADC Bank Device Driver provides an API to any ADC Bank of ADC pins in the target system. ADC requests are processed in order they are received and returned when completed.

4.2.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*RequestSingle)(
            void *aWorkspace,
            ADC_RequestSingle *aRequest);
} DEVICE_ADCBank;
```

4.2.2. RequestSingle()

Description:

Do a single ADC transfer. Will block until the A/D conversion is completed. Also protects multiple tasks from using the same resource.

Parameters:

void *aWorkspace – ADC Bank Device driver's workspace

ADC_RequestSingle *aRequest – ADC request and parameters.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.3 Backlight Device Driver API

The backlight device driver is used in conjunction with LCDs to provide light level control. Because many LCDs have their backlight in a separate hardware control, the control has been split out here. By separating the logic, the backlight control can provide an abstract design where 0 is off and 0xFFFF is 100% on without concern of the underlying hardware requirements.

4.3.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*On)(void *aWorkspace);
    T_uezError (*Off)(void *aWorkspace);
    T_uezError (*SetRatio)(
            void *aWorkspace,
            TUInt16 aRatio);
} DEVICE_Backlight;
```

4.3.2. On()

Description:

Turns on the backlight to the last set ratio level.

Parameters:

void *aWorkspace – Backlight device driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.3.3. Off()

Description:

Turns off the backlight.

Parameters:

void *aWorkspace – Backlight device driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.3.4. SetRatio()

Description:

Sets the ratio from 0 to 0xFFFF of the backlight setting when on. If currently off, this just sets the value to use when on. When on, this immediately changes the value.

Parameters:

void *aWorkspace – Backlight device driver's workspace

TUInt16 aRatio – 0 (darkest) to 0xFFFF (brightest)

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.4 Button Bank Device Driver API (DEVICE_ButtonBank)

The Button Bank Device Driver provides an API to group of buttons in a shared configuration. Buttons are typically grouped to allow easy reading of all the buttons at the same time. Buttons can also be disabled as no longer needed. The device driver only provides raw on/off functionality and does not provide queuing of events.

4.4.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*SetActiveButtons)(
            void *aWorkspace,
            TUInt32 aActiveButtons);
    T_uezError (*Read)(
            void *aWorkspace,
            TUInt32 *aButtons);
} DEVICE_ButtonBank;
```

4.4.2. SetActiveButtons()

Description:

Declare which of the buttons to be active and configure their hardware appropriately. Non active buttons will not report their state.

Parameters:

void *aWorkspace – Button Bank Device driver's workspace

TUInt32 aActiveButtons – A bit value of 1 indicates active, while 0 is inactive. Bits 0 to 31 represent the 32 buttons on this bank.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.4.3. Read()

Description:

Reads the raw state of all the active buttons at once. Bits with a value of 0 indicates the button is pushed while a value of 1 is a button not pushed.

Parameters:

void *aWorkspace – Button Bank Device driver's workspace

TUInt32 *aButtons – A bit value of 1 indicates not being pushed, while 0 indicates a button being pushed. Bits 0 to 31 represent the 32 buttons on this bank.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.5 EEPROM Device Driver API (DEVICE_EEPROM)

EEPROMs come in many different forms and flavors, but they have a common interface. Reading and writing to EEPROMs go through the DEVICE_EEPROM API. Because so many EEPROMs are very similar, a SetConfiguration command is used to declare the operating parameters for the EEPROM (write time, erase time, etc.).

4.5.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*SetConfiguration)(
            void *aWorkspace,
            T_EEPROMConfig *aConfig);
    T_uezError (*Read)(
            void *aWorkspace,
            TUInt32 aAddress,
            TUInt8 *aBuffer,
            TUInt32 aNumBytes);
    T_uezError (*Write)(
            void *aWorkspace,
            TUInt32 aAddress,
            TUInt8 *aBuffer,
            TUInt32 aNumBytes);
} DEVICE_EEPROM;
```

4.5.2. SetConfiguration()

Description:

Declares the page size, communication speed, and write times for the EEPROM. Standard EEPROMs are available in Types\EEPROM.h as macros with the EEPROM_CONFIG_ prefix.

Parameters:

void *aWorkspace – EEPROM device driver's workspace

T_EEPROMConfig *aConfig – Pointer to configuration information

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.5.3. Read()

Description:

Reads a series of bytes from the EEPROM at the given address.

Parameters:

void *aWorkspace – EEPROM device driver's workspace
TUInt32 aAddress – Offset into EEPROM with 0 being the base byte address.
TUInt8 *aBuffer – Buffer to receive bytes
TUInt32 aNumBytes – Number of bytes to read

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.5.4. Write()

Description:

Writes a series of bytes into the EEPROM at the given address.

Parameters:

void *aWorkspace – EEPROM device driver's workspace
TUInt32 aAddress – Offset into EEPROM with 0 being the base byte address.
TUInt8 *aBuffer – Buffer of bytes to write
TUInt32 aNumBytes – Number of bytes to write

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.6 Flash Device Driver API (DEVICE_Flash)

In addition to EEPROM memory, there is flash memory in the form of NOR flash memory or NAND flash memory. These memory devices must be erased in discrete blocks of bytes. The DEVICE_Flash API provides a method of reading, writing, and erasing blocks and the bytes within those blocks. In addition, information is returned about the structure of the memory as many devices can have multiple types of block sizes.

4.6.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*Open)(void *aWorkspace);
    T_uezError (*Close)(void *aWorkspace);
    T_uezError (*Read)(
            void *aWorkspace,
            TUInt32 aOffset,
            TUInt8 *aBuffer,
            TUInt32 aNumBytes);
    T_uezError (*Write)(
            void *aWorkspace,
            TUInt32 aOffset,
            TUInt8 *aBuffer,
            TUInt32 aNumBytes);
    T_uezError (*BlockErase)(
            void *aWorkspace,
            TUInt32 aOffset,
            TUInt32 aNumBytes);
    T_uezError (*ChipErase)(void *aWorkspace);
    T_uezError (*QueryReg)(
            void *aWorkspace,
            TUInt32 aReg,
            TUInt32 *aValue);
    T_uezError (*GetChipInfo)(
            void *aWorkspace,
            T_FlashChipInfo *aInfo);
    T_uezError (*GetBlockInfo)(
            void *aWorkspace,
            TUInt32 aOffset,
            T_FlashBlockInfo *aBlockInfo);
} DEVICE_Flash;
```

4.6.2. BlockErase()

Description:

Erase a range of blocks under the given range of bytes. This routine is aggressive and will erase a complete block if only a single byte touches the block. Therefore, knowledge of the blocks must be used to ensure that no more than expected is erased. See routine GetChipInfo for more details.

Parameters:

void *aWorkspace – Flash device driver's workspace
TUInt32 aOffset – Offset into flash device (0 address is start of device)
TUInt32 aNumBytes – Minimum number of bytes to erase

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.6.3. ChipErase()

Description:

Erase the complete flash device.

Parameters:

void *aWorkspace – Flash device driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.6.4. Close()

Description:

Close a previously opened flash device.

Parameters:

void *aWorkspace – Flash device driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.6.5. GetBlockInfo()

Description:

Returns information about the specific block under the given offset in the form of T_FlashBlockInfo

```
typedef struct {
   TUInt32 iOffset;
   TUInt32 iSize;
} T_FlashBlockInfo;
```

The iOffset is the block's starting byte offset and iSize is the size of the block (in bytes).

Parameters:

void *aWorkspace – Flash device driver's workspace
TUInt32 aOffset – Byte offset into flash device of block
T_FlashBlockInfo * aBlockInfo

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.6.6. GetChipInfo()

Description:

Return information about this Flash device. See T_FlashChipInfo below:

```
typedef struct {
   TUInt32 iNumBytesLow;
   TUInt32 iNumBytesHigh; // if bigger than 4 Gig
   TUInt8 iBitWidth;   // 8, 16, or 32

   TUInt32 iNumRegions;
   T_FlashChipRegion
         iRegions[FLASH_CHIP_MAX_TRACKED_REGIONS];
} T_FlashChipInfo;

typedef struct {
   TUInt32 iNumEraseBlocks;        // count, base 1
   TUInt32 iSizeEraseBlock;        // in bytes
} T_FlashChipRegion;
```

The T_FlashChipInfo structure contains a 64-bit size in bytes, the number of bits wide, and a number of region structures. A region is a sequential grouping of blocks of the same size. The flash device's offset starts at 0 and erase region occur one after another. For example, if a flash chip has 128 blocks of size 1024 bytes and then 8 blocks of size 256 bytes, the first region is at address 0x0, the second region is at address 0x20000, and the last byte is at 0x207FF.

Parameters:

void *aWorkspace – Flash device driver's workspace
T_FlashChipInfo *aInfo – Structure filled with information

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.6.7. Open()

Description:

Opens access to the Flash memory. Once open, the flash memory is expected only to be access with flash read and write commands and not accessed directly.

Parameters:

void *aWorkspace – Flash device driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.6.8. Read()

Description:

Safely read a number of bytes from the flash device to memory. If the flash device is not open, the flash device can be accessed directly.

Parameters:

void *aWorkspace – Flash device driver's workspace
TUInt32 aOffset – Offset into Flash with 0 being the base byte address.
TUInt8 *aBuffer – Place to store bytes read
TUInt32 aNumBytes – Number of bytes to read

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.6.9. QueryReg()

Description:

Read back the result of an internal register of the flash device (if available).

Parameters:

void *aWorkspace – Flash device driver's workspace
TUInt32 aReg – Register index to read
TUInt32 *aValue – Place to return result

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.6.10. Write()

Description:

Safely write a number of bytes from memory into the flash device.
NOTE: The flash memory should first be erased before writing. Reports an error if the memory cannot be written.

Parameters:

void *aWorkspace – Flash device driver's workspace
TUInt32 aOffset – Offset into Flash with 0 being the base byte address.
TUInt8 *aBuffer – Bytes to write
TUInt32 aNumBytes – Number of bytes to write

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

4.7 I2C Bus Device Driver API (DEVICE_I2C_BUS)

The I2C Bus Device Driver provides an API to any I2C Bus in the target system. I2C commands are put into request structures and processed one at a time by the system while the device driver ensures thread safety.

4.7.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*ProcessRequest)(void *aWorkspace, I2C_Request
*aRequest);
} DEVICE_I2C_BUS;
```

4.7.2. ProcessRequest()

Description:

Processes a single request of write and/or read commands in the I2C_Request structure.

Parameters:

void *aWorkspace – I2C Bus Device driver's workspace
I2C_Request *aRequest – I2C request with write and/or read command.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If a timeout occurs, returns UEZ_ERROR_TIMEOUT. If the device is not available or stops mid-read, returns UEZ_ERROR_NAK.

4.8 LCD Device Driver API (DEVICE_LCD)

The LCD Device Driver provides an API to any LCD screen's frame buffer as well as routines to control the brightness and if the LCD is on or off.

4.8.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*Open)(void *aWorkspace);
    T_uezError (*Close)(void *aWorkspace);
    T_uezError (*Configure)(
                    void *aWorkspace,
                    HAL_LCDController **aLCDController,
                    TUInt32 aBaseAddress,
                    DEVICE_Backlight **aBacklight);
    T_uezError (*GetInfo)(void *aWorkspace, T_uezLCDConfiguration
*aConfiguration);
    T_uezError (*GetFrame)(void *aWorkspace, TUInt32 aFrame, void
**aFrameBuffer);
    T_uezError (*ShowFrame)(void *aWorkspace, TUInt32 aFrame);
    T_uezError (*On)(void *aWorkspace);
    T_uezError (*Off)(void *aWorkspace);
    T_uezError (*Backlight)(void *aWorkspace, TUInt32 aLevel);
} DEVICE_LCD;
```

4.8.2. Backlight()

Description:

Turns on or off the backlight. A value of 0 is off and values of 1 or higher is higher levels of brightness (dependent on the LCD display). If a display is on/off only, this value will be 1 or 0 respectively.

Parameters:

void *aWorkspace – LCD Device driver's workspace
TUInt32 aLevel – Backlight intensity.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the device handle is bad, returns UEZ_ERROR_HANDLE_INVALID. If the backlight intensity level is invalid, returns UEZ_ERROR_OUT_OF_RANGE.

4.8.3. Close()

Description:

Close a previously open connection to the LCD screen, deactivating the display if this is the last instance.

Parameters:

void *aWorkspace – LCD Device driver's workspace

Results:

T_uezError – If closed, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.8.4. Configure()

Description:

Links the LCD device driver to a LCD Controller (if any needed).

Parameters:

void *aWorkspace – LCD Device driver's workspace
HAL_LCDController **aLCDController – LCD Controller HAL workspace to use.
TUInt32 aBaseAddress – Base memory address for all data.
DEVICE_Backlight **aBacklight – Backlight device driver (if any).

Results:

T_uezError – Return UEZ_ERROR_NONE.

4.8.5. GetFrame()

Description:

Returns a pointer to the frame memory in the LCD display.

Parameters:

void *aWorkspace – LCD Device driver's workspace
TUInt32 aFrame – Index to frame.
void **aFrameBuffer – Pointer to pointer to linear array of memory bytes of memory.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If an invalid device handle, returns UEZ_ERROR_HANDLE_INVALID. If the frame is incorrect, returns UEZ_ERROR_OUT_OF_RANGE.

4.8.6. GetInfo()

Description:

Links the LCD device driver to a LCD Controller (if any needed).

Parameters:

void *aWorkspace – LCD Device driver's workspace
T_uezLCDConfiguration *aConfiguration – The LCD's configuration information (x resolution, y resolution, color depth, number frames, etc.).

Results:

T_uezError – Return UEZ_ERROR_NONE.

4.8.7. Off()

Description:

Turns off the LCD display.

Parameters:

void *aWorkspace – LCD Device driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If an invalid device handle, returns UEZ_ERROR_HANDLE_INVALID.

4.8.8. On()

Description:

Turns on the LCD display.

Parameters:

void *aWorkspace – LCD Device driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If an invalid device handle, returns UEZ_ERROR_HANDLE_INVALID.

4.8.9. Open()

Description:

Open a connection to the LCD screen, starting it if this is the first connection.

Parameters:

void *aWorkspace – LCD Device driver's workspace

Results:

T_uezError – If opened, returns UEZ_ERROR_NONE. If cannot open, returns error code.

4.8.10. SetPaletteColor()

Description:

Change the color of a palette entry given the red, green, blue component of a particular color index. The color components are expressed in full 16-bit values at a higher resolution than the hardware can usually perform. The color is down shifted to what the hardware can handle.

Parameters:

void *aWorkspace – LCD Device driver's workspace
TUInt16 aRed – Red component (0 – 0xFFFF)
TUInt16 aGreen – Green component (0 – 0xFFFF)
TUInt16 aBlue – Blue component (0 – 0xFFFF)

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If an invalid device handle, returns UEZ_ERROR_HANDLE_INVALID. If the color index is incorrect, returns UEZ_ERROR_OUT_OF_RANGE.

4.8.11. ShowFrame()

Description:

Makes the passed frame the actively viewed frame on the LCD (if not already).

Parameters:

void *aWorkspace – LCD Device driver's workspace
TUInt32 aFrame – Index to frame.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If an invalid device handle, returns UEZ_ERROR_HANDLE_INVALID. If the frame is incorrect, returns UEZ_ERROR_OUT_OF_RANGE.

4.9 Mass Storage Device Driver API

The Mass Storage device driver is an interface to a device that stores any amount of data as an array of blocks. The interface is intended to be used with file systems to seek, read, and write blocks of data. Although typically used to represent flash cards and similar devices, it can be expanded for any other application (e.g. remote network block device, serial flash, RAM drive, etc.).

4.9.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*Init)(void *aWorkspace);
    T_uezError (*GetStatus)(void *aWorkspace, T_msStatus
      *aStatus);
    T_uezError (*Read)(
            void *aWorkspace,
            const TUInt32 aStart,
            const TUInt32 aNumBlocks,
            void *aBuffer);
    T_uezError (*Write)(
            void *aWorkspace,
            const TUInt32 aStart,
            const TUInt32 aNumBlocks,
            const void *aBuffer);
    T_uezError (*Sync)(void *aWorkspace);
    T_uezError (*GetSizeInfo)(void *aWorkspace, T_msSizeInfo
      *aInfo);
    T_uezError (*SetPower)(void *aWorkspace, TBool aOn);
    T_uezError (*SetLock)(void *aWorkspace, TBool aLock);
    T_uezError (*SetSoftwareWriteProtect)(void *aWorkspace, TBool
      aSWWriteProtect);
    T_uezError (*Eject)(void *aWorkspace);
    T_uezError (*MiscControl)(
            void *aWorkspace,
            TUInt32 aControlCode,
            void *aBuffer);
} DEVICE_MassStorage;
```

4.9.2. Init()

Description:

Initialize the Mass Storage device and confirm the existence of the support hardware (even if the medium is missing).

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code. Typical error codes include UEZ_ERROR_DEVICE_NOT_FOUND, UEZ_ERROR_TIMEOUT, and UEZ_ERROR_NOT_AVAILABE.

4.9.3. Eject()

Description:

For devices with medium that can be ejected, this function tries to mechanically eject the medium.

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code. Returns UEZ_ERROR_NOT_SUPPORTED if not a feature of the mass storage device.

4.9.4. GetSizeInfo()

Description:

Returns the number of sectors, size of individual sectors, and size of blocks (num sectors per block) of the target device.

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace

T_msSizeInfo *aInfo – return size info.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.9.5. GetStatus()

Description:

Returns T_msStatus, a list of bit fields:

MASS_STORAGE_STATUS_NEED_INIT – Need to call Init().

MASS_STORAGE_STATUS_NO_MEDIUM – Support hardware ready, but no mass storage medium has been found.

MASS_STORAGE_STATUS_WRITE_PROTECTED – Support hardware ready, mass storage medium exists, but medium is write protected (e.g. Secure SDCard with security enabled).

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace

T_msStatus *aStatus – Pointer to field to receive bit fields.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE with bit fields. Otherwise, returns error code.

4.9.6. MiscControl()

Description:

Because various unspecified controls may be needed not already specified in the Mass Control API, a MiscControl function has been provided. A control code and buffer is passed in. The buffer passed in can be for data going to the control code or being returned. This routine only provides pass through functionality.

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace
TUInt32 aControlCode – Device specific control code.
void *aBuffer – Data in and/or out for the control code.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE with bit fields. Otherwise, returns error code. UEZ_ERROR_ILLEGAL_OPERATION is returned for any unknown and unhandled aControlCode.

4.9.7. Read()

Description:

Reads one or more blocks at the given starting block number into a buffer. Data is read and stored sequentially.

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace
const TUInt32 aStart – Starting block number
const TUInt32 aNumBlocks – Number of blocks to read
void *aBuffer – pointer to buffer to receive data.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.9.8. SetLock()

Description:

For devices with medium that can be ejected, this function locks the medium from being removed. NOTE: This function may not always be implemented – even in devices that have the lock functionality.

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace
TBool aLock – ETrue if mass storage device is to be locked, else EFalse.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code. Returns UEZ_ERROR_NOT_SUPPORTED if not a feature of the mass storage device.

4.9.9. SetPower()

Description:

Turns on/off the target device for power saving. Not always implemented.

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace

TBool aOn – ETrue if mass storage device is to be on, else EFalse.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code. Returns UEZ_ERROR_NOT_SUPPORTED if not a feature of the mass storage device.

4.9.10. SetSoftwareWriteProtect()

Description:

The software write protect can be set here. Setting to a ETrue value causes writes to immediately fail without regard to the actual write protection status of the underlying mass storage device. This feature can be thought of a safety.

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace

TBool aSWWriteProtect – ETrue protects the device, EFalse removes the safety.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.9.11. Sync()

Description:

The sync routine is called to ensure that all transactions have completed and the device is ready for the next command. Primarily used to flush all write buffers.

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.9.12. Write()

Description:

Writes one or more blocks at the given starting block number using data from a buffer. Data is written from the buffer sequentially.

Parameters:

void *aWorkspace – Mass Storage Device driver's workspace
const TUInt32 aStart – Starting block number
const TUInt32 aNumBlocks – Number of blocks to write
void *aBuffer – pointer to buffer holding data to write.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.10 RTC Device Driver API

The RTC device driver provides routines to get and set the RTC's time.

4.10.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*Get)(void *aWorkspace, T_uezTimeDate *aTimeDate);
    T_uezError (*Set)(void *aWorkspace, const T_uezTimeDate
      *aTimeDate);
} DEVICE_RTC;
```

4.10.2. Get()

Description:

Get the current RTC clock reading.

Parameters:

void *aWorkspace – RTC Device driver's workspace
T_uezTimeDate *aTimeDate – Time and date returned.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.10.3. Set()

Description:

Set the current RTC clock.

Parameters:

void *aWorkspace – RTC Device driver's workspace
const T_uezTimeDate *aTimeDate – Time and date to set to

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.10.4. Validate()

Description:

Validate the current RTC date and time and if the date or time is invalid (typically to lost integrity from power out), reset to the given time and date. If no date and time is given, just report that the RTC is invalid.

Parameters:

void *aWorkspace – RTC Device driver's workspace
const T_uezTimeDate *aTimeDate – Time and date to set to if invalid. If 0 is passed, the time and date is not changed.

Results:

T_uezError – If valid, returns UEZ_ERROR_NONE. If invalid and no given date and time, returns UEZ_ERROR_INVALID. If invalid and date and time given, returns same as Set() function (UEZ_ERROR_NONE or error code when set).

4.10.5. SetClockOutHz()

Description:

Some RTCs have the ability to toggle an output control line. This routine enables the output and sets the rate to the closest match.

Parameters:

void *aWorkspace – RTC Device driver's workspace

TUInt32 aHertz – Speed in Hz (usual values of 1, 1024, 32768 are used). A speed of 0 turns off the feature.

Results:

T_uezError – If set, returns UEZ_ERROR_NONE. If unknown speed, returns UEZ_ERROR_NOT_SUPPORTED if aHertz is too big or too small. Other low level errors might also be reported.

4.11 SPI Bus Device Driver API

The SPI Bus system currently uses a simple polled method of doing transfers. An SPI request is passed into the device driver, the data is transferred doing both send and receive simultaneously. When complete, control returns to the caller.

Future versions will incorporate interrupts and queuing to allow an SPI task to be processed in the background without polling.

4.11.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*TransferPolled)(void *aWorkspace, SPI_Request
      *aRequest);

    void (*Start)(void *aWorkspace, SPI_Request *aRequest);
    TUInt32 (*TransferInOut)(
                    void *aWorkspace,
                    SPI_Request *aRequest,
                    TUInt32 aNum);
    void (*Stop)(void *aWorkspace, SPI_Request *aRequest);
} DEVICE_SPI_BUS;
```

4.11.2. TransferPolled()

Description:

Do a single SPI transfer using polling methods. Will block current task until the SPI is completed.

Parameters:

void *aWorkspace – SPI Bus Device driver's workspace

SPI_Request *aRequest – Parameters for SPI transfer including GPIO for CS, speed of transfer, number of transfers, size of transfers, and polarity for chip select and clock lines.
NOTE: aRequest->iDataIn points to data to go into the SPI device and aRequest->iDataOut points to data that comes out of the SPI device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.11.3. Start()

Description:

Start a piecewise SPI transaction. Stop() must be called at some point to end the transfer.

Parameters:

void *aWorkspace – SPI Bus Device driver's workspace

SPI_Request *aRequest – Parameters for SPI transfer including GPIO for CS, speed of transfer, size of transfers, and polarity for chip select and clock lines. Number of transfers is not used in this structure.

Results:

None.

4.11.4. Stop()

Description:

End a piecewise SPI transaction.

Parameters:

void *aWorkspace – SPI Bus Device driver's workspace

SPI_Request *aRequest – Parameters for SPI transfer including GPIO for CS, speed of transfer, size of transfers, and polarity for chip select and clock lines. Number of transfers is not used in this structure.

Results:

None.

4.11.5. TransferInOut()

Description:

Do multiple transactions out the SPI while also reading in 1 for 1.

Parameters:

void *aWorkspace – SPI Bus Device driver's workspace

SPI_Request *aRequest – Parameters for SPI transfer including GPIO for CS, speed of transfer, size of transfers, and polarity for chip select and clock lines. Number of transfers is not used in this structure.

TUInt32 aNum – number of transactions to perform.

NOTE: aRequest->iDataIn points to data to go into the SPI device and aRequest->iDataOut points to data that comes out of the SPI device.

Results:

TUInt32 – Number of transactions performed.

4.12 SSP Bus Device Driver API

The SSP Bus system currently uses a simple polled method of doing transfers. An SSP request is passed into the device driver, the data is transferred doing both send and receive simultaneously. When complete, control returns to the caller.

Future versions will incorporate interrupts and queuing to allow an SSP task to be processed in the background without polling.

4.12.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*TransferPolled)(void *aWorkspace, SSP_Request
*aRequest);
} DEVICE_SSP_BUS;
```

4.12.2. TransferPolled()

Description:

Do a single SSP transfer using polling methods. Will block current task until the transfer is complete.

Parameters:

void *aWorkspace – SPI Bus Device driver's workspace

SSP_Request *aRequest – Parameters for SSP transfer including GPIO for CS, speed of transfer, number of transfers, size of transfers, and polarity for chip select and clock lines.
NOTE: aRequest->iDataIn points to data to go into the SPI device and aRequest->iDataOut points to data that comes out of the SPI device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.13 Stream Device Driver API

Stream drivers provide access to any device that sends and receives characters. FIFOs are used to hold characters waiting to go out or waiting to be received.

4.13.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*Open)(void *aWorkspace);
    T_uezError (*Close)(void *aWorkspace);
    T_uezError (*Control)(void *aWorkspace, TUInt32 aControl, void
*aControlData);
    T_uezError (*Read)(
            void *aWorkspace,
            TUInt8 *aData,
            TUInt32 aNumBytes,
            TUInt32 *aNumBytesRead,
            TUInt32 aTimeout);
    T_uezError (*Write)(
            void *aWorkspace,
            TUInt8 *aData,
            TUInt32 aNumBytes,
            TUInt32 *aNumBytesWritten,
            TUInt32 aTimeout);
} DEVICE_STREAM;
```

4.13.2. Command()

Description:

Passes a low level command to the underlying driver handling the stream data.

Parameters:

void *aWorkspace – Stream Device Driver's workspace
TUInt32 aCommand – Command number to execute
void *aCommandData – Pointer to data to pass into the workspace and/or data to return.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.13.3. Close()

Description:

Closes the stream and deactivates it if the last close.

Parameters:

void *aWorkspace – Stream Device Driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.13.4. Open()

Description:

Opens the stream and activates it if the first time opened.

Parameters:

void *aWorkspace – Stream Device Driver's workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.13.5. Read()

Description:

Read data from the stream. A timeout between successive characters is given. If a timeout occurs, the number of bytes sent is reported.

Parameters:

void *aWorkspace – Stream Device Driver's workspace

TUInt8 *aData – Place to store data

TUInt32 aNumBytes – Number of bytes to read

TUInt32 *aNumBytes – Number of bytes actually read. If a timeout occurs, this value is less than aNumBytes. A value of 0 is allowed to not return the number.

Results:

T_uezError – Error code. UEZ_ERROR_TIMEOUT is returned if timeout occurs trying to read the full amount and not enough data is provided. UEZ_ERROR_NONE is reported if all data is returned.

4.13.6. Write()

Description:

Write data out the stream. A timeout between successive characters is given. If a timeout occurs, the number of bytes written is reported.

Parameters:

void *aWorkspace – Stream Device Driver's workspace

TUInt8 *aData – Pointer to data to send

TUInt32 aNumBytes – Number of bytes to write

TUInt32 *aNumBytes – Number of bytes actually written. If a timeout occurs, this value is less than aNumBytes. A value of 0 is allowed to not return the number.

Results:

T_uezError – Error code. UEZ_ERROR_TIMEOUT is returned if timeout occurs trying to read the full amount and not enough data is provided. UEZ_ERROR_NONE is reported if all data is returned.

4.13.7. Flush()

Description:

Flush data in a stream out the underlying device waiting until the flush is complete.

Parameters:

void *aWorkspace – Stream Device Driver's workspace

Results:

T_uezError – Error code. UEZ_ERROR_NONE is reported if completed successfully.

4.14 Temperature Device Driver API (DEVICE_Temperature)

Various temperature readings can be provided by one or more temperature devices. The API provides an abstract format for access any temperature device.

4.14.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Read the temperature in 15.16 signed fixed point format
    //  degrees Celsius.
    T_uezError (*Read)(
            void *aWorkspace,
            TInt32 *aTemperature);
} DEVICE_Temperature;
```

4.14.2. Read()

Description:

Make a single immediate read of the temperature.

Parameters:

void *aWorkspace – Temperature device driver's workspace

TInt32 *aTemperature – Temperature in signed 15.16 fixed point format (1 bit sign, 15 bit signed integer, 16 bit fraction). Temperature usually is Celsius (but may change depending on device).

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code

4.15 Touch Screen Device Driver API

The touch screen device driver provides an interface to touch screen readings, settings, and calibration. The primary goal of the driver is to provide calibrated readings when polled.

4.15.1. Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    T_uezError (*Open)(void *aWorkspace);
    T_uezError (*Close)(void *aWorkspace);
    T_uezError (*Poll)(void *aWorkspace, T_uezTSReading
*aReading);

    T_uezError (*CalibrationStart)(void *aWorkspace);
    T_uezError (*CalibrationAddReading)(
                    void *aWorkspace,
                    const T_uezTSReading *aReadingTaken,
                    const T_uezTSReading *aReadingExpected);
    T_uezError (*CalibrationEnd)(void *aWorkspace);
    T_uezError (*SetTouchDetectSensitivity)(
                    void *aWorkspace,
                    TUInt16 aLowLevel,
                    TUInt16 aHighLevel);
} DEVICE_TOUCHSCREEN ;
```

4.15.2. Close()

Description:

Closes the touch screen access.

Parameters:

void *aWorkspace – Touch Screen Device Driver workspace

Results:

T_uezError – If the device is closed, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.15.3. Poll()

Description:

Forces the touch screen to poll for an immediate reading.

Parameters:

void *aWorkspace – Touch Screen Device Driver workspace
T_uezTSReading *aReading – Touch screen reading

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Returns UEZ_ERROR_TIMEOUT if the hardware is unresponsive.

4.15.4. Open()

Description:

Opens access to the touch screen.

Parameters:

void *aWorkspace – Touch Screen Device Driver workspace

Results:

T_uezError – If the device is opened, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.15.5. CalibrationStart()

Description:

Puts the touch screen device driver in calibration mode.

Parameters:

void *aWorkspace – Touch Screen Device Driver workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns error code.

4.15.6. CalibrationAddReading()

Description:

Adds a data point to the touch screen calibration.

Parameters:

void *aWorkspace – Touch Screen Device Driver workspace

T_uezTSReading *aReadingTaken – Reading taken using UEZTSGetReading.

T_uezTSReading *aReadingExpected – Reading that was expected (ideal).

Results:

T_uezError – If done (last calibration point), returns UEZ_ERROR_NONE. If more data is needed, returns UEZ_ERROR_NOT_ENOUGH_DATA.

4.15.7. CalibrationEnd()

Description:

Finishes the calibration and uses it.

Parameters:

void *aWorkspace – Touch Screen Device Driver workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the calibration was terminated without enough data points, returns UEZ_ERROR_NOT_ENOUGH_DATA.

4.15.8. SetTouchDetectSensitivity()

Description:

Change the low (press) and high (release) points of the touch detection code. These points are hardware specific, but the values passed are 0-0xFFFF as a scale between 0 V and reference Voltage. Notice that putting a gap between high and low provides a range where the press/release is reported as the previous press/release until the threshold is met.

Parameters:

T_uezDevice aDevice – Previously opened touch screen device.
TUInt16 aLowLevel – Low (press) reference level (0-0xFFFF)
TUInt16 aHighLevel – High (press) reference level (0-0xFFFF)

Results:

If successful, return UEZ_ERROR_NONE, else returns error code.

4.16 USB Device API

The USB Device provides an interface to the USB Device hardware that synchronizes commands to the low level hardware with the RTOS.

4.16.1. USB Device Driver Definition

```
typedef struct {
    // Header
    T_uezDeviceInterface iDevice;

    // Functions
    // Functions
    T_uezError (*Configure)(
        void *aWorkspace,
        const TUInt8 *aDescriptorTable);
    void (*Initialize)(void *aWorkspace);
    void (*Connect)(void *aWorkspace);
    void (*Disconnect)(void *aWorkspace);
    void (*SetAddress)(void *aWorkspace, TUInt8 aAddress);
    void (*EndpointConfigure)(
            void *aWorkspace,
            TUInt8 aEndpoint,
            TUInt16 aMaxPacketSize);
    TInt16 (*Read)(
            void *aWorkspace,
            TUInt8 aEndpoint,
            TUInt8 *aData,
            TUInt16 iMaxLen);
    TUInt16 (*Write)(
            void *aWorkspace,
            TUInt8 aEndpoint,
            TUInt8 *aData,
            TUInt16 aLength);
    void (*SetStallState)(
            void *aWorkspace,
            TUInt8 aEndpoint,
            TBool aStallState);
    TBool (*IsStalled)(
            void *aWorkspace,
            TUInt8 aEndpoint);
    void (*EndpointInterruptsEnable)(
            void *aWorkspace,
            TUInt8 aEndpoint);
    void (*EndpointInterruptsDisable)(
            void *aWorkspace,
            TUInt8 aEndpoint);
    void (*ForceAddressAndConfigurationNum)(
            void *aWorkspace,
            TUInt8 aAddress,
            TUInt8 aConfigNum);
    T_uezError (*RegisterEndpointCallback)(
            void *aWorkspace,
            TUInt8 aEndpoint,
            T_USBEndpointHandlerFunc aEndpointFunc);
    T_uezError (*UnregisterEndpointCallback)(
            void *aWorkspace,
            TUInt8 aEndpoint);
    T_uezError (*RegisterRequestTypeCallback)(
            void *aWorkspace,
            TUInt8 aType,
            void *aBuffer,
            void *aCallbackWorkspace,
            T_USBRequestHandlerFunc aRequestHandler);
    T_uezError (*UnregisterRequestTypeCallback)(
            void *aWorkspace,
            TUInt8 aType);
    T_uezError (*ProcessEndpoints)(void *aWorkspace, TUInt32
                  aTimeout);
    void (*InterruptNakEnable)(void *aWorkspace, TUInt8 aNakBits);
} DEVICE_USB_DEVICE ;
```

4.16.2. Configure()

Description:

Configure and setup the USB Device with the given settings.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace

T_USBDevSettings *aSettings – Particular configuration for this USB Device controller. Contains a pointer to the descriptor table and a pointer to the USB Device configuration structure.

Results:

None

4.16.3. Initialize()

Description:

Initialize the hardware (based on the previously given configuration). All endpoints are realized and enabled. Only need to connect next.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace

Results:

None

4.16.4. Connect()

Description:

Connect the USB device to the USB device. Up til this point, the USB Device has not been available to the USB Host.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace

Results:

None

4.16.5. Disconnect()

Description:

Remove the USB Device from the USB bus and make it disappear from the USB Host.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace

Results:

None

4.16.6. SetAddress()

Description:

Set the address of this USB Device on the USB bus. Usually the USB Host will tell when to change this address.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace

TUInt8 aAddress – 7 bit address of this device.

Results:

None

4.16.7. SetConfiguration()

Description:

A USB Device can have multiple configurations but always at least one. The USB Host will signal when to change configurations and which configuration to use. This routine configures the hardware on which configuration to use. A configuration index of 0 will cause the device to return to a non-configured state.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace

TUInt8 aConfigurationIndex – Index of configuration or 0 for no configuration.

Results:

None

4.16.8. GetConfiguration()

Description:

Returns the current configuration index. If returns 0, the USB device is not configured.

Parameters:
void *aWorkspace – USB Device Controller Driver workspace

Results:
TUInt8– Index of configuration or 0 for no configuration.

4.16.9. EndpointConfigure()

Description:
Realizes and enables an endpoint.
Parameters:
void *aWorkspace – USB Device Controller Driver workspace
TUInt8 aEndpoint – Endpoint to realize. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.
Results:
None.

4.16.10. Read()

Description:
Read data from an endpoint and then clears the read buffer.
NOTE: All data available should be read from the endpoint by calling this routine once.
Parameters:
void *aWorkspace – USB Device Controller Driver workspace
TUInt8 aEndpoint – Endpoint to read from. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.
TUInt8 *aData – Place to store data read.
TUInt16 iMaxLen – Maximum number of bytes that can be read.
Results:
TInt16 – Number of bytes read. If data is not valid, returns -1.

4.16.11. Write()

Description:
Write data to an endpoint and then validate (mark ready to send) the write buffer.
NOTE: All data to be sent should be written to the endpoint by calling this routine once.
Parameters:
void *aWorkspace – USB Device Controller Driver workspace
TUInt8 aEndpoint – Endpoint to write to. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.
TUInt8 *aData – Data to write.
TUInt16 aLength – Number of bytes to write.
Results:
TInt16 – Number of bytes actually written.

4.16.12. SetStallState()

Description:

Set the stall state of an endpoint.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace

TUInt8 aEndpoint – Endpoint to set stall state. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

TBool aStallState – ETrue for stall, else EFalse for not stalled.

Results:

None.

4.16.13. IsStalled()

Description:

Determine if an endpoint is stalled.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace

TUInt8 aEndpoint – Endpoint possibly stalled. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

Results:

TBool – ETrue if stalled, else EFalse.

4.16.14. EndpointInterruptsEnable()

Description:

Allow an endpoint to generate interrupts to the interrupt handler routine.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace

TUInt8 aEndpoint – Endpoint to enable endpoint interrupts. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

Results:

None.

4.16.15. EndpointInterruptsDisable()

Description:

Disallow an endpoint to generate interrupts to the interrupt handler routine.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace

TUInt8 aEndpoint – Endpoint to disable endpoint interrupts. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

Results:

None.

4.16.16. ForceAddressAndConfiguration()

Description:

Forces the USB device to go to the given address number and configuration number.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace
TUInt8 aConfigNum – Index of configuration or 0 for no configuration.
TUInt8 aAddress – 7 bit address of this device.

Results:

None.

4.16.17. RegisterEndpointCallback()

Description:

Register an endpoint callback routine that is called when data is ready/requested on the given endpoint and ProcessEndpoints is called.

Parameters:

void *aWorkspace – USB Device Controller Driver workspace
TUInt8 aEndpoint – Endpoint to do callback. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.
T_USBEndpointHandlerFunc aEndpointFunc – Pointer to function to call when data is ready and ProcessEndpoints is called.

Results:

T_uezError – Error code UEZ_ERROR_NOT_AVAILABLE is returned if endpoint is already registered, or UEZ_ERROR_NONE if successful.

4.16.18. UnregisterEndpointCallback()

Description:

Unregister an endpoint so the previously associated callback routine is no longer called when endpoint data was ready.

Parameters:

void *aWorkspace – USB Device Driver workspace
TUInt8 aEndpoint – Endpoint with callbacks. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

Results:

T_uezError – Error code UEZ_ERROR_NOT_AVAILABLE is returned if endpoint is already registered, or UEZ_ERROR_NONE if successful.

4.16.19. RegisterRequestTypeCallback()

Description:

Register a handler for different types of control 0 requests.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aType – Type of control 0 request. Can be
USB_REQUEST_TYPE_STANDARD,
USB_REQUEST_TYPE_CLASS,
USB_REQUEST_TYPE_VENDOR, or
USB_REQUEST_TYPE_RESERVED.

void *aBuffer – Buffer to received data (up to endpoint max sized packet).

T_USBRequestHandlerFunc aRequestHandler – Pointer to function to call when endpoint 0 receives data of this type.

Results:

T_uezError – Returns UEZ_ERROR_NONE if successfully registered.
Returns UEZ_ERROR_NOT_AVAILABLE if slot already registered.
Returns UEZ_ERROR_ILLEGAL_OPERATION if incorrect type.

4.16.20. UnregisterRequestTypeCallback()

Description:

Unregister a handler for different types of control 0 requests.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aType – Type of control 0 request. Can be
USB_REQUEST_TYPE_STANDARD,
USB_REQUEST_TYPE_CLASS,
USB_REQUEST_TYPE_VENDOR, or
USB_REQUEST_TYPE_RESERVED.

Results:

T_uezError – Returns UEZ_ERROR_NONE if successfully unregistered.
Returns UEZ_ERROR_NOT_AVAILABLE if slot already unregistered.
Returns UEZ_ERROR_ILLEGAL_OPERATION if incorrect type.

4.16.21. InterruptNakEnable()

Description:

Declares which type of Naks to generate interrupts.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aNakBits – bit combination of the following:

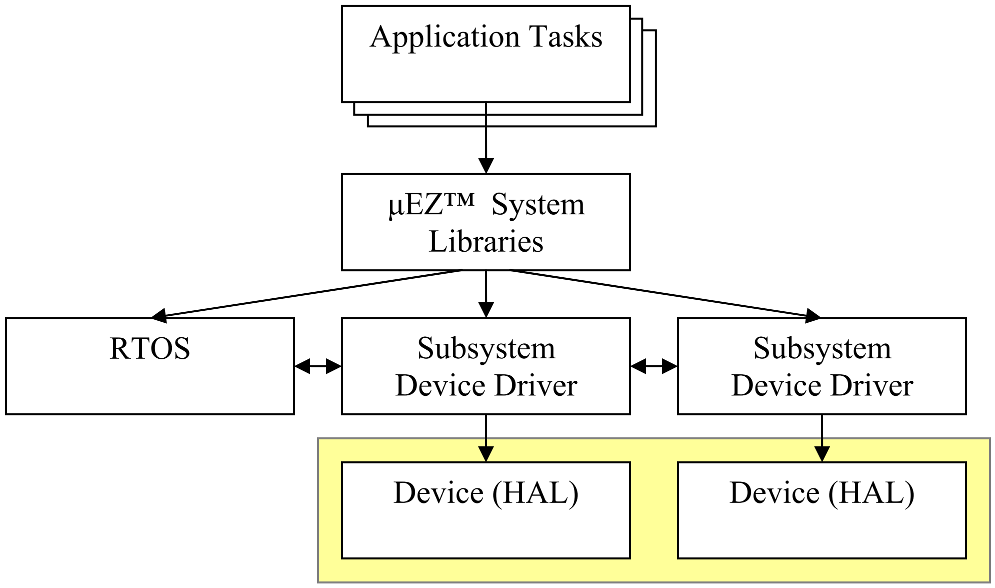
USB_DEVICE_SET_MODE_AP_CLK, _INAK_CI, INAK_CO, INAK_II, _INAK_IO, _INAK_BI, _INAK_BO, where C = command, I=Interrupt, and B= Bulk.

Results:

T_uezError – Returns UEZ_ERROR_NONE if successfully unregistered. Returns UEZ_ERROR_NOT_AVAILABLE if slot already unregistered. Returns UEZ_ERROR_ILLEGAL_OPERATION if incorrect type.

5.0 HAL Layer

The HAL Layer is the layer of routines that tie directly to the hardware. Routines at this layer do not know about the upper layers of the application, libraries, or even RTOS. Semaphores, queues, and other RTOS constructs are handled by the Subsystem Device Driver layer. At the HAL layer, each function assumes it is the only code running on select section of the hardware.



Families of HAL interface types are defined in the directory “Include/HAL”. For example, a project may be using a serial UART on the processor and a serial UART on the base board. Both of these are serial UARTs and their HAL drivers will use the HAL_Serial interface (see “Include/HAL/Serial.h”).

Each HAL device must provide a HAL interface. The HAL interface is a structure registered at power on and starts with a T_halInterface substructure. T_halInterface is as follows:

```
typedef struct {
    const char *iName;
    TUInt16 iVersion;   // Version is 0x103 for version 1.03
    T_uezError (*iInitializeWorkspace)(void *aWorkspace);
    TUInt32 iWorkspaceSize;
} T_halInterface;
```

Each field starts with a pointer to a unique name. Names are given in the following format “<device type>:<company>:<hardware>”. For example “Touchscreen:TI:TSC2046” means it is a touch screen device using Texas Instrument’s TSC2046 integrated circuit.

Next comes the 16-bit version number. The upper byte is the major version and provides compatibility information – the number must match to be compatible. The lower byte is the minor version. The minor version must be equal or greater to the required minor version to be compatible. For example, version 0x0203 is needed for a project. If the HAL driver is 0x0300 or 0x0100, the major version is different and definitely not compatible. A HAL driver of version 0x0200 is also not compatible because it has a lower minor version. However, a HAL driver of version 0x0204 is compatible even though it may have a few other extra features.

The fields iInitializeWorkspace and iWorkspaceSize are used when the HAL driver is registered. A workspace is allocated on the heap and calls iInitializeWorkspace to setup the driver.

The rest of the HAL interface structure is typically a list of function pointers that expose the features of the HAL device (although other fields can be provided as required by the standard interface).

5.1 HAL Registration

When a processor or platform initializes, the HAL interface is registered to the uEZ system and a workspace instance created. By registering, other devices can find the system using a string name that matches the one used at registration. By registering unique devices to generic devices names (e.g., use "SPI0" for "SPI:NXP:LPC2478:SPI0"), the rest of the system can quickly map to the hardware without knowledge of the particular hardware type being used.

5.1.1. HALInterfaceRegister ()

Description:

Registers a new HAL interface and sets up a workspace (instance) for it.

Parameters:

const char * const aName – Pointer to name to register the device as.
T_halInterface *aInterface – Pointer to HAL interface.
T_uezInterface *aInterfaceHandle – Pointer to newly created interface handle.
T_halWorkspace **aWorkspace – Returns a pointer to the newly created and initialized workspace.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If memory for the workspace cannot be allocated, returns UEZ_ERROR_OUT_OF_MEMORY. If there is no more space in the HAL registration table, returns UEZ_ERROR_OUT_OF_HANDLES. If initialization of the workspace fails, an error specific to that HAL may be also returned.

5.1.2. HALInterfaceGetWorkspace()

Description:

Using a HAL's interface handle, returns a pointer to the HAL instance's workspace.

Parameters:

T_uezInterface *aInterfaceHandle – Pointer to interface handle.
T_halWorkspace **aWorkspace – Returns a pointer to the workspace.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.1.3. HALInterfaceFind()

Description:

Searches for a HAL instance by name.

Parameters:

const char * const aName – Name to search for.

T_halWorkspace **aWorkspace – Returns a pointer to the workspace if found.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns UEZ_ERROR_NOT_FOUND.

5.2 ADC API

Analog-to-Digital Converters (ADC) pins are used to convert voltage levels to a range of values. ADC pins are configured in groups called ADC Banks. Each bank shares the same clock and timing configuration. The ADC API allows sampling of each ADC pin individually by passing in a ADC Request. When the sampling is complete, a callback routine is called.

5.2.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*Configure)(
            void *aWorkspace,
            void *aCallbackWorkspace,
            const T_adcBankCallbackInterface * const
            aCallbackAPI);
    T_uezError (*RequestSingle)(
            void *aWorkspace,
            ADC_RequestSingle *aRequest);
} HAL_ADCBank;
```

5.2.2. Configure()

Description:

Finishes the calibration and uses it.

Parameters:

void *aWorkspace – ADCBank HAL Driver workspace

void *aCallbackWorkspace – Workspace to pass into callback routine.

T_adcBankCallbackInterface *aCallbackAPI – structure containing list of callback routines.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.2.3. RequestSingle()

Description:

Configures and samples on a single ADC pin within the ADC Bank. Returns immediately, but calls callback routine given in Configure() when complete.

Parameters:

void *aWorkspace – ADCBank HAL Driver workspace

ADC_RequestSingle *aRequest – ADC sampling parameters.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.3 BatteryRAM API

Processors that have battery backed ram can use the BatteryRAM API to help read, write, and verify the data integrity.

5.3.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*Read)(void *aWorkspace, const TUInt32 aOffset,
            TUInt8 *aData);
    T_uezError (*Write)(void *aWorkspace, const TUInt32 aOffset,
            const TUInt8 aData);
    TBool (*Valid)(void *aWorkspace);
    TUInt32 (*GetSize)(void *aWorkspace);
} HAL_BatteryRAM;
```

5.3.2. GetSize()

Description:

Returns the size in bytes of the battery backed RAM (minus any over head used by this driver).

Parameters:

void *aWorkspace – BatteryRAM HAL Driver workspace

Results:

TUInt32 – Number of bytes available for data starting at offset 0.

5.3.3. IsValid()

Description:

Returns flag telling if the battery backed RAM is valid.

Parameters:

void *aWorkspace – BatteryRAM HAL Driver workspace

Results:

TBool – ETrue if battery backed RAM appears valid, else EFalse.

5.3.4. Read()

Description:

Reads a single byte out of the battery backed RAM device.

Parameters:

void *aWorkspace – BatteryRAM HAL Driver workspace
const TUInt32 aOffset – Byte offset into battery RAM
TUInt8 *aData – Place to store returned byte

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code. If offset is out of range, returns UEZ_ERROR_ILLEGAL_PARAMETER.

5.3.5. Write()

Description:

Writes a single byte into the battery backed RAM device.

Parameters:

void *aWorkspace – BatteryRAM HAL Driver workspace
const TUInt32 aOffset – Byte offset into battery RAM
const TUInt8 aData – byte of data to write

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If out of range, returns UEZ_ERROR_ILLEGAL_PARAMETER. Otherwise, returns an error code.

5.4 DAC API

Digital-to-Analog Converters (DAC) pins are used to output voltage levels over a range of values.

5.4.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*Read)(void *aWorkspace, TUInt32 * value);
    T_uezError (*ReadMilliVolts)(void *aWorkspace, TUInt32 *
            milliVolt);
    T_uezError (*Write)(void *aWorkspace, const TUInt32 value);
    T_uezError (*WriteMilliVolts)(void *aWorkspace, const TUInt32
            milliVolt);
    T_uezError (*SetBias)(void *aWorkspace, TBool bias);
    T_uezError (*SetVRef)(void *aWorkspace, const TUInt32 VRef);
} HAL_DAC;
```

5.4.2. Read()

Description:

Reads the current raw output value from the DAC. Requires specific knowledge of the underlying DAC.

Parameters:

void *aWorkspace – DAC HAL Driver workspace
TUInt32 *aValue – Place to store returned value

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.4.3. ReadMilliVolts()

Description:

Returns the current output value from the DAC in millivolts.

Parameters:

void *aWorkspace – DAC HAL Driver workspace
const TUInt32 *aMilliVolts – Millivolt reading

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.4.4. Write()

Description:

Writes the raw output value to the DAC. Requires specific knowledge of the underlying DAC.

Parameters:

void *aWorkspace – DAC HAL Driver workspace
TUInt32 aValue – Raw value to use.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.4.5. WriteMilliVolts()

Description:

Writes the output value in millivolts to the DAC.

Parameters:

void *aWorkspace – DAC HAL Driver workspace
const TUInt32 aMilliVolts – Millivolt value to set

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.4.6. SetBias()

Description:

Turns on or off the bias.

Parameters:

void *aWorkspace – DAC HAL Driver workspace
TBool aBias – ETrue to turn on bias, else EFalse to turn off.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.4.7. SetVref()

Description:

Set the voltage reference value in millivolts.

Parameters:

void *aWorkspace – DAC HAL Driver workspace
const TUInt32 aVRef – Voltage reference in millivolts.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.5 EMAC API

The Ethernet MAC (Media Access Controller) configures and interacts with the hardware to send and receive Ethernet frames on a network.

5.5.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*Configure)(void *aWorkspace, T_EMACSettings
            *iSettings);
    TBool (*CheckFrameReceived)(void *aWorkspace);
    TUInt16 (*StartReadFrame)(void *aWorkspace);
    void (*EndReadFrame)(void *aWorkspace);
    void (*RequestSend)(void *aWorkspace);
    void (*DoSend)(void *aWorkspace, TUInt16 aFrameSize);
    void (*CopyToFrame)(
                void *aWorkspace,
                void *aSource,
                TUInt32 aSize);
    void (*CopyFromFrame)(
                void *aWorkspace,
                void *aDestination,
                TUInt32 aSize);
} HAL_EMAC;
```

5.5.2. CheckFrameReceived()

Description:

Determine if a frame has been received and is awaiting processing.

Parameters:

void *aWorkspace – EMAC HAL driver workspace

Results:

TBool – ETrue if frame received, else EFalse.

5.5.3. Configure()

Description:

Configure and setup the EMAC with settings controlling the PHY and the MAC address. The PHY is initialized at this point.

Parameters:

void *aWorkspace – EMAC HAL driver workspace

T_EMACSettings *aSettings – structure with PHY type and MAC address.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.5.4. CopyToFrame()

Description:

Copies bytes from MCU-memory to frame port building upon the existing frame bytes.

Parameters:

void *aWorkspace – EMAC HAL driver workspace
void *aSource – Location of data to send.
TUInt32 aSize – Number of bytes to send.

Results:

None

5.5.5. CopyFromFrame()

Description:

Copies bytes from frame port to MCU-memory.

Parameters:

void *aWorkspace – EMAC HAL driver workspace
void *aDestination – Location to store data. NOTE: location must start at word boundary.
TUInt32 aSize – Number of bytes to store.

Results:

None

5.5.6. DoSend()

Description:

The frame being sent out is complete and needs only to be sent.

Parameters:

void *aWorkspace – EMAC HAL driver workspace
TUInt16 aFrameSize – Complete size of this frame.

Results:

None.

5.5.7. EndReadFrame()

Description:

Ends reading of the EMAC frame and releases it.

Parameters:

void *aWorkspace – EMAC HAL driver workspace

Results:

None

5.5.8. RequestSend()

Description:

Requests space in the EMAC memory for sending an outgoing frame.

NOTE: Assumes there is space available and does not return an error.

Parameters:

void *aWorkspace – EMAC HAL driver workspace

Results:

None.

5.5.9. StartReadFrame()

Description:

Reads the length of the received Ethernet frame and returns.

Parameters:

void *aWorkspace – EMAC HAL driver workspace

Results:

TUInt16 – Length of currently received Ethernet frame.

5.6 GPIO HAL API

A General Purpose Input/Output (GPIO) port is defined to be a group of up to 32 pins. Each pin can be individually changed to a high or low state, input or output mode, and the pins can be polled for high/low readings. The GPIO HAL goes a step further by masking off certain pins that must be protected.

5.6.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*Configure)(void *aWorkspace, TUInt32
aUsablePins);
    T_uezError (*Activate)(void *aWorkspace, TUInt32 aPortPins);
    T_uezError (*Deactivate)(void *aWorkspace, TUInt32 aPortPins);
    T_uezError (*SetOutputMode)(void *aWorkspace, TUInt32
aPortPins);
    T_uezError (*SetInputMode)(void *aWorkspace, TUInt32
aPortPins);
    T_uezError (*Set)(void *aWorkspace, TUInt32 aPortPins);
    T_uezError (*Clear)(void *aWorkspace, TUInt32 aPortPins);
    T_uezError (*Read)(void *aWorkspace, TUInt32 aPortPins,
TUInt32 *aPinsRead);
} HAL_GPIOPort ;
```

5.6.2. Configure()

Description:

Finishes the calibration and uses it.

Parameters:

void *aWorkspace – GPIO HAL Driver workspace

TUInt32 aUsablePins – Bit mask of which pins are allowed to be used on this port.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.6.3. Activate()

Description:

Configures the given pins to be used for GPIO. Defaults all pins to input mode.

Parameters:

void *aWorkspace – GPIO HAL Driver workspace

TUInt32 aPortPins – Bit mask of which pins are to be activated for GPIO.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.6.4. Deactivate()

Description:

Releases the pins from GPIO mode.

Parameters:

void *aWorkspace – GPIO HAL Driver workspace

TUInt32 aPortPins – Bit mask of which pins are to be released.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.6.5. SetOutputMode()

Description:

Configures active pins for output mode.

Parameters:

void *aWorkspace – GPIO HAL Driver workspace

TUInt32 aPortPins – Bit mask of which pins are to be output mode.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.6.6. SetInputMode()

Description:

Configures active pins for input mode.

Parameters:

void *aWorkspace – GPIO HAL Driver workspace

TUInt32 aPortPins – Bit mask of which pins are to be input mode.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.6.7. Set()

Description:

Configures active output pins for HIGH (set) output.

Parameters:

void *aWorkspace – GPIO HAL Driver workspace
TUInt32 aPortPins – Bit mask of which output pins are to be set high.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.6.8. Clear()

Description:

Configures active output pins for LOW (clear) output.

Parameters:

void *aWorkspace – GPIO HAL Driver workspace
TUInt32 aPortPins – Bit mask of which output pins are to be set low.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.6.9. Read()

Description:

Reads group of GPIO pins. Returns pattern of bits where 1 is HIGH and 0 is LOW.

Parameters:

void *aWorkspace – GPIO HAL Driver workspace
TUInt32 aPortPins – Bit mask of which pins to read.
TUInt32 *aPinsRead – Pointer to place to store results.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.7 I2C BUS HAL API

The I2C Bus HAL API defines the most basic functions for doing single read or write I2C transactions. Requests are started, processed via interrupts, and a callback routine is called upon completion.

5.7.1. HAL Driver Definition

```
typedef typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    void (*RequestRead)(
        void *aWorkspace,
        I2C_Request *iRequest,
        void *aCallbackWorkspace,
        I2CRequestCompleteCallback aCallbackFunc);
    void (*RequestWrite)(
        void *aWorkspace,
        I2C_Request *iRequest,
        void *aCallbackWorkspace,
        I2CRequestCompleteCallback aCallbackFunc);
} HAL_I2CBus;
```

5.7.2. RequestRead()

Description:

Starts performing an I2C read with the read parameters given in the I2C_Request structure. When complete, it will call the callback function with given workspace.

Parameters:

void *aWorkspace – I2C Bus HAL Driver workspace

I2C_Request *iRequest – Request structure with read parameters.

void *aCallbackWorkspace – Workspace given to callback function.

I2CRequestCompleteCallback aCallbackFunc – Callback function to call upon completion.

Results:

None

5.7.3. RequestWrite()

Description:

Starts performing an I2C write with the write parameters given in the I2C_Request structure. When complete, it will call the callback function with given workspace.

Parameters:

void *aWorkspace – I2C Bus HAL Driver workspace
I2C_Request *iRequest – Request structure with write parameters.
void *aCallbackWorkspace – Workspace given to callback function.
I2CRequestCompleteCallback aCallbackFunc – Callback function to call upon completion.

Results:

None

5.8 LCD Controller HAL API

A LCD Controller is the hardware that controls the signals to a LCD screen. Configuring these signals changes based on LCD screen type and the particular LCD controller used. The LCD Controller HAL attempts to create an abstract model for setting up and controlling the LCD Controller. Usually a higher level LCD Device Driver interfaces the LCD Controller to a specific LCD panel and hides the implementation details from the application.

5.8.1. HAL Driver Definition

```
typedef struct {
    // Common header
    T_halInterface iInterface;

    TUInt32 iFeatures;

    // Functions
    T_uezError (*Configure)(void *aWorkspace,
T_LCDControllerSettings *aSettings);
    T_uezError (*On)(void *aWorkspace);
    T_uezError (*Off)(void *aWorkspace);
    T_uezError (*SetBacklightLevel)(void *aWorkspace, TUInt32
aLevel);
    T_uezError (*SetBaseAddr)(void *aWorkspace, TUInt32
aBaseAddress, TBool aSync);
    T_uezError (*SetPaletteColor)(
                    void *aWorkspace,
                    TUInt32 aColorIndex,
                    TUInt16 aRed,
                    TUInt16 aGreen,
                    TUInt16 aBlue);
} HAL_LCDController;
```

5.8.2. Configure()

Description:

Sets up the parameters for the LCD controller.

Parameters:

void *aWorkspace – LCD Controller HAL Driver workspace

T_LCDControllerSettings *aSettings – Specific LCD Controller operating parameters.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.8.3. Off()

Description:

Turns off the LCD controller's output.

Parameters:

void *aWorkspace – LCD Controller HAL Driver workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.8.4. On()

Description:

Turns on the LCD controller's output.

Parameters:

void *aWorkspace – LCD Controller HAL Driver workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.8.5. SetBacklightLevel()

Description:

If the LCD Controller has the ability to control the backlight, set the level here.

Parameters:

void *aWorkspace – LCD Controller HAL Driver workspace
TUInt32 aLevel – Light level from 0 (min) to 0xFFFFFFFF (max)

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.8.6. SetBaseAddr()

Description:

Specifies the base address of the pixels/characters for the LCD Display. The sync option tells if the update should be synchronized with the next raster scan of the controller.

Parameters:

void *aWorkspace – LCD Controller HAL Driver workspace
TUInt32 aBaseAddress – Base address of pixels/characters.
TBool aSync – ETrue if update is to be synchronized with raster, else EFalse.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the address is out of range, returns UEZ_ERROR_OUT_OF_RANGE. Otherwise, returns an error code.

5.8.7. SetPaletteColor()

Description:

Change the color of a palette entry given the red, green, blue component of a particular color index. The color components are expressed in full 16-bit values at a higher resolution than the hardware can usually perform. The color is down shifted to what the hardware can handle.

Parameters:

void *aWorkspace – LCD Controller HAL Driver workspace
TUInt32 aColorIndex – Index in the palette.
TUInt16 aRed – Red component (0 – 0xFFFF)
TUInt16 aGreen – Green component (0 – 0xFFFF)
TUInt16 aBlue – Blue component (0 – 0xFFFF)

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. If the color index is out of range, returns UEZ_ERROR_OUT_OF_RANGE. Otherwise, returns an error code.
.

5.9 Pulse Width Modulation (PWM) HAL API

Modern processors have Pulse Width Modulation outputs that are put together in a group of match registers controlled by a single master match register. The PWM HAL API provides a common interface for easy configuration of multiple PWMs with different trigger sources.

5.9.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    // Sets the clock rate using match register 0
    T_uezError (*SetMaster)(
        void *aWorkspace,
        TUInt32 aPeriod,
        TUInt8 aMatchRegister);
    T_uezError (*SetMatchRegister)(
        void *aWorkspace,
        TUInt8 aMatchRegister,
        TUInt32 aMatchPoint,
        TBool aDoInterrupt,
        TBool aDoCounterReset,
        TBool aDoStop);
    T_uezError (*EnableSingleEdgeOutput)(
        void *aWorkspace,
        TUInt8 aMatchRegister);
    T_uezError (*DisableOutput)(
        void *aWorkspace,
        TUInt8 aMatchRegister);
} HAL_PWM;
```

5.9.2. SetMaster()

Description:

Declares the number of cycles in the PWM master and which register to use as the master. The number of cycles is specific to the hardware (usually in number of CPU cycles).

Parameters:

void *aWorkspace – PWM HAL workspace
TUInt32 aPeriod – Number of cycles for one complete master cycle.
TUInt8 aMatchRegister – Which of the match register is the master.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.9.3. SetMatchRegister()

Description:

Configure a single match register

Parameters:

void *aWorkspace – PWM HAL workspace
TUInt8 aMatchRegister – Which of the match registers to configure.
TUInt32 aMatchPoint – Number of cycles until a match occurs
TBool aDoInterrupt – ETrue if an interrupt should be generated, else EFalse.
TBool aDoCounterReset – Etrue if the master counter should be reset to zero on a match.
TBool aDoStop – ETrue if the master counter should be stopped on a match. Use SetMaster to reconfigure and restart the master.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.9.4. EnableSingleEdgeOutput()

Description:

Configure a single match register to output on its matching output pin. Output is high until it matches, then goes low, then goes back to high when the master counter resets to zero.

Parameters:

void *aWorkspace – PWM HAL workspace
TUInt8 aMatchRegister – Which match register should be put in single edge output mode.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.9.5. DisableOutput()

Description:

Stops any output on the corresponding match register's output pin and keeps output high.

Parameters:

void *aWorkspace – PWM HAL workspace
TUInt8 aMatchRegister – Which match register should be disabled.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code

5.10 RTC HAL API

For devices that have a very low level API for their RTC (such as a RTC built on the processor), an API for getting and setting the time and date has been provided.

5.10.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*Get)(void *aWorkspace, T_uezTimeDate *aTimeDate);
    T_uezError (*Set)(void *aWorkspace, const T_uezTimeDate
      *aTimeDate);
} HAL_RTC;
```

5.10.2. Get()

Description:

Get the current date and time.

Parameters:

void *aWorkspace –RTC HAL Driver workspace

T_uezTimeDate *aTimeDate – Time and date of the current time.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.10.3. Set()

Description:

Set the current date and time.

Parameters:

void *aWorkspace –RTC HAL Driver workspace

const T_uezTimeDate *aTimeDate – Time and date of the current time.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.11 Serial HAL API

The Serial HAL provides serial style (UART/modem) device access. The HAL itself only handles the processing of individual characters, but leaves the buffering to a higher level Stream Device Driver.

5.11.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*Configure)(
                void *aWorkspace,
                void *aCallbackWorkspace,
                HAL_Serial_Callback_Received_Byte
                  aReceivedByteCallback,
                HAL_Serial_Callback_Transmit_Empty
                  aTransmitEmptyCallback);
    T_uezError (*SetSerialSettings)(
                    void *aWorkspace,
                    T_Serial_Settings *aSettings);
    T_uezError (*GetSerialSettings)(
                    void *aWorkspace,
                    T_Serial_Settings *aSettings);
    T_uezError (*Activate)(void *aWorkspace);
    T_uezError (*Deactivate)(void *aWorkspace);
    T_uezError (*OutputByte)(
                void *aWorkspace,
                TUInt8 aByte);
    T_uezError (*SetHandshakingControl)(
                    void *aWorkspace,
                    T_serialHandshakingControl aControl);
    T_uezError (*GetHandshakingStatus)(
                    void *aWorkspace,
                    T_serialHandshakingStatus *aControl);
} HAL_Serial ;
```

5.11.2. Activate()

Description:

Enable processing of characters on this serial device.

Parameters:

void *aWorkspace – Serial HAL Driver workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.11.3. Configure()

Description:

Sets up the device with two callback routines (one for send buffer is empty, one for when receive buffer has data) and a pointer to the callback's workspace.

Parameters:

void *aWorkspace – Serial HAL Driver workspace.

void *aCallbackWorkspace – Workspace of callback.

HAL_Serial_Callback_Received_Byte aReceivedByteCallback – Callback for when byte received.

HAL_Serial_Callback_Transmit_Empty aTransmitEmptyCallback – Callback when output buffer has become empty.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.11.4. Deactivate()

Description:

Disables processing of characters on this serial device.

Parameters:

void *aWorkspace – Serial HAL Driver workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.11.5. GetSerialSetting()

Description:

Gets the serial configuration of the device.

Parameters:

void *aWorkspace – Serial HAL Driver workspace.

T_Serial_Settings *aSettings – Serial speed and mode of operation.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.11.6. OutputByte()

Description:

Output a single byte to the serial device.

Parameters:

void *aWorkspace – Serial HAL Driver workspace.

TUInt8 aByte – Byte to output.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.11.7. SetSerialSetting()

Description:

Sets the serial configuration of the device.

Parameters:

void *aWorkspace – Serial HAL Driver workspace.

T_Serial_Settings *aSettings – Serial speed and mode of operation.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.12 SPI HAL API

The SPI HAL takes single SPI requests, processes them, and then returns the result.

5.12.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*TransferPolled)(void *aWorkspace, SPI_Request
               *aRequest);

    void (*Start)(void *aWorkspace, SPI_Request *aRequest);
    TUInt32 (*TransferInOut)(
                void *aWorkspace,
                SPI_Request *aRequest,
                TUInt32 aSend);
    void (*Stop)(void *aWorkspace, SPI_Request *aRequest);
} HAL_SPI;
```

5.12.2. TransferPolled()

Description:

Do a SPI transfer (both read and write) out of the SPI port using the given request.

NOTE: Does not use interrupts and waits in loop while processing.

Parameters:

void *aWorkspace – SPI HAL Driver workspace

SPI_Request *aRequest – Parameters for SPI transfer including GPIO for CS, speed of transfer, number of transfers, size of transfers, and polarity for chip select and clock lines.
NOTE: aRequest->iDataIn points to data to go into the SPI device and aRequest->iDataOut points to data that comes out of the SPI device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.12.3. Start()

Description:

Start a piecewise SPI transaction. Stop() must be called at some point to end the transfer.

Parameters:

void *aWorkspace – SPI HAL Driver workspace

SPI_Request *aRequest – Parameters for SPI transfer including GPIO for CS, speed of transfer, size of transfers, and polarity for chip select and clock lines. Number of transfers is not used in this structure.

Results:

None.

5.12.4. Stop()

Description:

End a piecewise SPI transaction.

Parameters:

void *aWorkspace – SPI HAL Driver workspace

SPI_Request *aRequest – Parameters for SPI transfer including GPIO for CS, speed of transfer, size of transfers, and polarity for chip select and clock lines. Number of transfers is not used in this structure.

Results:

None.

5.12.5. TransferInOut()

Description:

Do multiple transactions out the SPI while also reading in 1 for 1.

Parameters:

void *aWorkspace – SPI HAL Driver workspace

SPI_Request *aRequest – Parameters for SPI transfer including GPIO for CS, speed of transfer, size of transfers, and polarity for chip select and clock lines. Number of transfers is not used in this structure. NOTE: aRequest->iDataIn points to data to go into the SPI device and aRequest->iDataOut points to data that comes out of the SPI device.

TUInt32 aNum – number of transactions to perform.

Results:

TUInt32 – Number of transactions performed.

5.13 SSP HAL API

The SSP HAL takes single SSP requests, processes them, and then returns the result.

5.13.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*TransferPolled)(void *aWorkspace, SSP_Request
                  *aRequest);
} HAL_SPI;
```

5.13.2. TransferPolled()

Description:

Do a SSP transfer (both read and write) out of the SSP port using the given request.

NOTE: Does not use interrupts and waits in loop while processing.

Parameters:

void *aWorkspace – SSP HAL Driver workspace

SSP_Request *aRequest – Parameters for SSP transfer including GPIO for CS, speed of transfer, number of transfers, size of transfers, and polarity for chip select and clock lines.

NOTE: aRequest->iDataIn points to data to go into the SPI device and aRequest->iDataOut points to data that comes out of the SPI device.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.14 Timer HAL API

All timers share the Timer HAL API. Timers can be used to generate interrupts or control external hardware.

5.14.1. HAL Driver Definition

```
typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    // Sets the clock rate using match register 0
    T_uezError (*SetTimerMode)(
        void *aWorkspace,
        T_Timer_Mode aMode);
    T_uezError (*SetCaptureSource)(
        void *aWorkspace,
        TUInt32 aSourceIndex);
    T_uezError (*SetMatchRegister)(
        void *aWorkspace,
        TUInt8 aMatchRegister,
        TUInt32 aMatchPoint,
        TBool aDoInterrupt,
        TBool aDoCounterReset,
        TBool aDoStop);
    T_uezError (*SetExternalControl)(
        void *aWorkspace,
        TUInt8 aMatchRegister,
        T_Timer_ExternalControl aExtControl);
    void (*Reset)(void *aWorkspace);
    void (*Enable)(void *aWorkspace);
    void (*Disable)(void *aWorkspace);
} HAL_Timer;
```

5.14.2. Disable()

Description:

Disable the timer and stop running. Does not reset the counter.

Parameters:

void *aWorkspace – Timer HAL driver workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.14.3. Enable()

Description:

Enables the timer and continues running it from where it last stopped.

Parameters:

void *aWorkspace – Timer HAL driver workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.14.4. Reset()

Description:

Reset the timer and start back at 0.

Parameters:

void *aWorkspace – Timer HAL driver workspace

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.14.5. SetTimerMode()

Description:

Declare the mode the timer will be operating in.

Parameters:

void *aWorkspace – Timer HAL driver workspace

T_Timer_Mode aMode – Timer mode (internal clock, rising capture, falling capture, or both).

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.14.6. SetCaptureSource()

Description:

When the timer is set to one of the capture modes, declare the index of the source. The index is target specific.

Parameters:

void *aWorkspace – Timer HAL driver workspace

TUInt32 aSourceIndex – Capture source for this timer.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.14.7. SetMatchRegister()

Description:

Setup the timer's operations on a match and when it matches.

Parameters:

void *aWorkspace – Timer HAL driver workspace

TUInt8 aMatchRegister – Which of the several match registers assigned to this timer.

TUInt32 aMatchPoint – What the timer has to reach in order to match.

TBool aDoInterrupt – ETrue if an interrupt is to be fired off on a match, else EFalse.

TBool aDoCounterReset – ETrue if the counter for the timer needs to be reset on a match, else EFalse.

TBool aDoStop – ETrue if the counter is to be stopped on a match, else EFalse.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.14.8. SetExternalControl()

Description:

Setup how a match register controls the output match pin and when (if any at all).

Parameters:

void *aWorkspace – Timer HAL driver workspace

TUInt8 aMatchRegister – Match register of timer and corresponding external pin.

T_Timer_ExternalControl aExtControl – How the external pin is affected (none, clear, set, or toggle) when timer reaches match point of match register.

Results:

T_uezError – If successful, returns UEZ_ERROR_NONE. Otherwise, returns an error code.

5.15 USB Device Controller API

The USB Device controller provides an interface to the USB device hardware, control of endpoints, and descriptor table interface.

5.15.1. HAL Driver Definition

```
typedef struct {
    const TUInt8 *iUSBDescriptorTable;
    T_USBDevice *iUSBDevice;
} T_USBDeviceControllerSettings;

typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*Configure)(
        void *aWorkspace,
        T_USBDeviceControllerSettings *iSettings);
    void (*Initialize)(void *aWorkspace);
    void (*Connect)(void *aWorkspace);
    void (*Disconnect)(void *aWorkspace);
    void (*SetAddress)(void *aWorkspace, TUInt8 aAddress);
    void (*SetConfiguration)(
                void *aWorkspace,
                TUInt8 aConfigurationIndex);
    TUInt8 (*GetConfiguration)(void *aWorkspace);
    void (*EndpointConfigure)(
            void *aWorkspace,
            TUInt8 aEndpoint,
            TUInt16 aMaxPacketSize);
    TInt16 (*Read)(
            void *aWorkspace,
            TUInt8 aEndpoint,
            TUInt8 *aData,
            TUInt16 iMaxLen);
    TUInt16 (*Write)(
            void *aWorkspace,
            TUInt8 aEndpoint,
            TUInt8 *aData,
            TUInt16 aLength);
    void (*SetStallState)(
            void *aWorkspace,
            TUInt8 aEndpoint,
            TBool aStallState);
    TBool (*IsStalled)(
            void *aWorkspace,
            TUInt8 aEndpoint);
    void (*EndpointInterruptsEnable)(
            void *aWorkspace,
            TUInt8 aEndpoint);
    void (*EndpointInterruptsDisable)(
            void *aWorkspace,
            TUInt8 aEndpoint);
    void (*ForceAddressAndConfigurationNum)(
            void *aWorkspace,
            TUInt8 aAddress,
            TUInt8 aConfigNum);
    void (*ProcessEndpoints)(void *aWorkspace);
    const TUInt8 * (*GetDescriptor)(
            void *aWorkspace,
            TUInt8 aType,
            TUInt16 aIndex,
            TUInt16 *aLength);
    void (*InterruptNakEnable)(void *aWorkspace, TUInt8 aNakBits);
} HAL_USBDeviceController ;
```

5.15.2. Configure()

Description:

Configure and setup the USB Device with the given settings.

Parameters:

void *aWorkspace – USB Device Driver workspace

T_USBDevSettings *aSettings – Particular configuration for this USB Device controller. Contains a pointer to the descriptor table and a pointer to the USB Device configuration structure.

Results:

None

5.15.3. Initialize()

Description:

Initialize the hardware (based on the previously given configuration). All endpoints are realized and enabled. Only need to connect next.

Parameters:

void *aWorkspace – USB Device Driver workspace

Results:

None

5.15.4. Connect()

Description:

Connect the USB device to the USB device. Up til this point, the USB Device has not been available to the USB Host.

Parameters:

void *aWorkspace – USB Device Driver workspace

Results:

None

5.15.5. Disconnect()

Description:

Remove the USB Device from the USB bus and make it disappear from the USB Host.

Parameters:

void *aWorkspace – USB Device Driver workspace

Results:

None

5.15.6. SetAddress()

Description:

Set the address of this USB Device on the USB bus. Usually the USB Host will tell when to change this address.

Parameters:

void *aWorkspace – USB Device Driver workspace
TUInt8 aAddress – 7 bit address of this device.

Results:

None

5.15.7. SetConfiguration()

Description:

A USB Device can have multiple configurations but always at least one. The USB Host will signal when to change configurations and which configuration to use. This routine configures the hardware on which configuration to use. A configuration index of 0 will cause the device to return to a non-configured state.

Parameters:

void *aWorkspace – USB Device Driver workspace
TUInt8 aConfigurationIndex – Index of configuration or 0 for no configuration.

Results:

None

5.15.8. GetConfiguration()

Description:

Returns the current configuration index. If returns 0, the USB device is not configured.

Parameters:

void *aWorkspace – USB Device Driver workspace

Results:

TUInt8– Index of configuration or 0 for no configuration.

5.15.9. EndpointConfigure()

Description:

Realizes and enables an endpoint.

Parameters:

void *aWorkspace – USB Device Driver workspace
TUInt8 aEndpoint – Endpoint to realize. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

Results:

None.

5.15.10. Read()

Description:

Read data from an endpoint and then clears the read buffer.
NOTE: All data available should be read from the endpoint by calling this routine once.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aEndpoint – Endpoint to read from. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

TUInt8 *aData – Place to store data read.

TUInt16 iMaxLen – Maximum number of bytes that can be read.

Results:

TInt16 – Number of bytes read. If data is not valid, returns -1.

5.15.11. Write()

Description:

Write data to an endpoint and then validate (mark ready to send) the write buffer.

NOTE: All data to be sent should be written to the endpoint by calling this routine once.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aEndpoint – Endpoint to write to. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

TUInt8 *aData – Data to write.

TUInt16 aLength – Number of bytes to write.

Results:

TInt16 – Number of bytes actually written.

5.15.12. SetStallState()

Description:

Set the stall state of an endpoint.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aEndpoint – Endpoint to set stall state. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

TBool aStallState – ETrue for stall, else EFalse for not stalled.

Results:

None.

5.15.13. IsStalled()

Description:

Determine if an endpoint is stalled.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aEndpoint – Endpoint possibly stalled. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

Results:

TBool – ETrue if stalled, else EFalse.

5.15.14. EndpointInterruptsEnable()

Description:

Allow an endpoint to generate interrupts to the interrupt handler routine.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aEndpoint – Endpoint to enable endpoint interrupts. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

Results:

None.

5.15.15. EndpointInterruptsDisable()

Description:

Disallow an endpoint to generate interrupts to the interrupt handler routine.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aEndpoint – Endpoint to disable endpoint interrupts. Endpoints are 0-15 with bit 7 set if IN direction, or bit 7 is clear for OUT direction.

Results:

None.

5.15.16. ForceAddressAndConfiguration()

Description:

Forces the USB device to go to the given address number and configuration number.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aConfigNum – Index of configuration or 0 for no configuration.

TUInt8 aAddress – 7 bit address of this device.

Results:

None.

5.15.17. GetDescriptor()

Description:

Search for a descriptor in the descriptor table based on the type and the index (of that type). Return a pointer to the data and fill in the length of the data.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aType – Type of descriptor to find

TUInt16 aIndex – Index to descriptor (zero based)

TUInt16 *aLength – Pointer to place to store length

Results:

TUInt8 * -- Pointer to descriptor, or 0 for none.

5.15.18. ProcessEndpoints()

Description:

Go through all endpoints and determine if an endpoint is ready and if it is, call the function associated with that endpoint (see USB_Device). Clear any related interrupts.

Parameters:

void *aWorkspace – USB Device Driver workspace

Results:

None.

5.15.19. InterruptNakEnable()

Description:

Declares which type of Naks to generate interrupts.

Parameters:

void *aWorkspace – USB Device Driver workspace

TUInt8 aNakBits – bit combination of the following:

USB_DEVICE_SET_MODE_AP_CLK, _INAK_CI, INAK_CO, INAK_II, _INAK_IO, _INAK_BI, _INAK_BO, where C = command, I=Interrupt, and B= Bulk.

Results:

T_uezError – Returns UEZ_ERROR_NONE if successfully unregistered. Returns UEZ_ERROR_NOT_AVAILABLE if slot already unregistered. Returns UEZ_ERROR_ILLEGAL_OPERATION if incorrect type.

5.16 USB Host API

The USB Host provides an interface to one or more USB devices (although typically a single one). The API provides routines to configure endpoints, get descriptors, get strings, send/receive control packets, send/receive bulk packets, and send/receive interrupt packets. Currently no support for isochronomous transactions have been provided.

NOTE: Some implementation only support one device address and should only use a device address of 1 in these cases.

5.16.1. HAL Driver Definition

```
typedef struct {
    void (*Connected)(void *aCallbackWorkspace);
    void (*Disconnected)(void *aCallbackWorkspace);
} T_USBHostCallbacks;

typedef struct {
    // Header
    T_halInterface iInterface;

    // Functions
    T_uezError (*SetCallbacks)(
        void *aWorkspace,
        void *aCallbackWorkspace,
        const T_USBHostCallbacks *aCallbackAPI);
    T_uezError (*Init)(void *aWorkspace);
    void (*ResetPort)(void *aWorkspace);
    void * (*AllocBuffer)(void *aWorkspace, TUInt32 aSize);
    T_uezError (*ProcessTD)(
        void *aWorkspace,
        volatile HCED *aEndDesc,
        TUInt32 aToken,
        volatile TUInt8 *buffer,
        TUInt32 buffer_len,
        TUInt32 aTimeout);
    T_usbHostDeviceState (*GetState)(void *aWorkspace);

    T_uezError (*SetControlPacketSize)(
            void *aWorkspace,
            TUInt8 aDeviceAddress,
            TUInt16 aPacketSize);
    T_uezError (*SetAddress)(
            void *aWorkspace,
            TUInt8 aAddress);
    T_uezError (*ConfigureEndpoint)(
            void *aWorkspace,
            TUInt8 aDeviceAddress,
            TUInt8 aEndpointAndInOut,   // bit 0x80 is set if IN
            TUInt16 aMaxPacketSize);
    T_uezError (*SetConfiguration)(
            void *aWorkspace,
            TUInt8 aDeviceAddress,
            TUInt32 aConfiguration);

    T_uezError (*Control)(
            void *aWorkspace,
            TUInt8 aDeviceAddress,
            TUInt8 aBMRequestType,
            TUInt8 aRequest,
            TUInt16 aValue,
            TUInt16 aIndex,
            TUInt16 aLength,
            void *aBuffer,
            TUInt32 aTimeout);
    T_uezError (*GetDescriptor)(
            void *aWorkspace,
            TUInt8 aDeviceAddress,
            TUInt8 aType,
            TUInt8 aIndex,
            TUInt8 *aBuffer,
            TUInt32 aSize,
            TUInt32 aTimeout);
    T_uezError (*GetString)(
            void *aWorkspace,
            TUInt8 aDeviceAddress,
            TUInt32 aIndex,
            TUInt8 *aBuffer,
            TUInt32 aSize,
            TUInt32 aTimeout);
    T_uezError (*BulkOut)(
            void *aWorkspace,
            TUInt8 aDeviceAddress,
            TUInt8 aEndpoint,
            TUInt8 *aBuffer,
            TUInt32 aSize,
            TUInt32 aTimeout);
    T_uezError (*BulkIn)(
            void *aWorkspace,
            TUInt8 aDeviceAddress,
            TUInt8 aEndpoint,
            TUInt8 *aBuffer,
            TUInt32 aSize,
            TUInt32 aTimeout);
    T_uezError (*InterruptOut)(
            void *aWorkspace,
            TUInt8 aDeviceAddress,
            TUInt8 aEndpoint,
            TUInt8 *aBuffer,
            TUInt32 aSize,
            TUInt32 aTimeout);
    T_uezError (*InterruptIn)(
            void *aWorkspace,
            TUInt8 aDeviceAddress,
            TUInt8 aEndpoint,
            TUInt8 *aBuffer,
            TUInt32 aSize,
            TUInt32 aTimeout);
} HAL_USBHost;
```

5.16.2. Init()

Description:

Initialize the USB Host hardware, reset all structures, and setup interrupts.

Parameters:

void *aWorkspace – USB Host driver workspace

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code.

5.16.3. AllocBuffer()

Description:

Allocate a buffer in the USB Host's memory space to transfer data. The USB Host system always moves data in and out of these buffers to avoid copying (i.e. zero copy).

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt32 aSize – Size of buffer to allocate.

Results:

void * -- Pointer to created buffer, or 0 if no buffer could be created.

5.16.4. BulkIn()

Description:

Receive a bulk packet from the given endpoint.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aDeviceAddress – USB device to provide bulk packet
TUInt8 aEndpoint – Endpoint to get bulk packet from.
void *aBuffer – zero-copy buffer previous created with AllocBuffer()
TUInt32 aSize – maximum size of buffer
TUInt32 aTimeout – Length of time before aborted.

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code. Returns UEZ_ERROR_TIMEOUT if timeout occurs.

5.16.5. BulkOut()

Description:

Send a bulk packet to the given endpoint.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aDeviceAddress – USB device to receive bulk packet
TUInt8 aEndpoint – Endpoint to send bulk packet to.
void *aBuffer – zero-copy buffer previous created with AllocBuffer()
TUInt32 aSize – maximum size of buffer
TUInt32 aTimeout – Length of time before aborted.

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code. Returns UEZ_ERROR_TIMEOUT if timeout occurs.

5.16.6. ConfigureEndpoint()

Description:

Setup an endpoints maximum packet size to a particular device address.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aDeviceAddress – Which device to affect.
TUInt8 aEndpointAndInOut – endpoint number plus 0x80 if IN direction or 0x00 if OUT direction.
TUInt32 aSize – Size of buffer to allocate.

Results:

void * -- Pointer to created buffer, or 0 if no buffer could be created.

5.16.7. Control()

Description:

Send a control packet to the device. If aLength is non-zero, data is returned in the given buffer.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aDeviceAddress – USB device to receive control packet
TUInt8 aBMRequestType – Request type to send (usually USB_HOST_TO_DEVICE | USB_REQ_TYPE_CLASS | USB_RECIPIENT_INTERFACE)
TUInt8 aRequest – Request (control code) to send.
TUInt16 aValue – Value parameter being sent
TUInt16 aIndex – Index parameter being sent
TUInt16 aLength – Length of return data.
void *aBuffer – zero-copy buffer previous created with AllocBuffer()
TUInt32 aTimeout – Length of time before aborted.

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code. Returns UEZ_ERROR_TIMEOUT if timeout occurs.

5.16.8. GetDescriptor()

Description:

Fetch descriptor from target device.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aDeviceAddress – USB device to receive control packet
TUInt8 aType – Type of descriptor (see USB_DESCRIPTOR_TYPE_xxx).
TUInt16 aIndex – Index of descriptor to find
void *aBuffer – zero-copy buffer previous created with AllocBuffer()
TUInt32 aSize – maximum size of buffer
TUInt32 aTimeout – Length of time before aborted.

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code. Returns UEZ_ERROR_TIMEOUT if timeout occurs.

5.16.9. GetState()

Description:

Read the state of the USB Host. It can be USB_HOST_STATE_CLEAR (never connected), USB_HOST_STATE_CONNECTED (device is plugged in), or USB_HOST_STATE_DISCONNECTED (device was removed).

Parameters:

void *aWorkspace – USB Host driver workspace

Results:

T_usbHostDeviceState – State of USB Host.

5.16.10. GetString()

Description:

Fetch string from target device.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aDeviceAddress – USB device to receive control packet
TUInt16 aIndex – Index of descriptor to find
void *aBuffer – zero-copy buffer previous created with AllocBuffer()
TUInt32 aSize – maximum size of buffer
TUInt32 aTimeout – Length of time before aborted.

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code. Returns UEZ_ERROR_TIMEOUT if timeout occurs.

5.16.11. InterruptIn()

Description:

Receive an interrupt packet from the given endpoint.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aDeviceAddress – USB device to provide interrupt packet
TUInt8 aEndpoint – Endpoint to get interrupt packet from.
void *aBuffer – zero-copy buffer previous created with AllocBuffer()
TUInt32 aSize – maximum size of buffer
TUInt32 aTimeout – Length of time before aborted.

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code. Returns UEZ_ERROR_TIMEOUT if timeout occurs.

5.16.12. InterruptOut()

Description:

Send an interrupt packet to the given endpoint.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aDeviceAddress – USB device to receive interrupt packet
TUInt8 aEndpoint – Endpoint to send interrupt packet to.
void *aBuffer – zero-copy buffer previous created with AllocBuffer()
TUInt32 aSize – maximum size of buffer
TUInt32 aTimeout – Length of time before aborted.

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code. Returns UEZ_ERROR_TIMEOUT if timeout occurs.

5.16.13. ProcessTD ()

DEPRECATED

5.16.14. ResetPort()

Description:

Send a reset command to the USB Host port. This typically is done immediately after a connection has been detected.

Parameters:

void *aWorkspace – USB Host driver workspace

Results:

None.

5.16.15. SetAddress()

Description:

Configures the current device waiting for a device address. If the first device, recommend using address 1.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aAddress – USB device address for last connected device to become.

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code.

5.16.16. SetCallbacks ()

Description:

Link the USB Host hardware to the given callbacks. It is assumed these callbacks occur within interrupt routines.

Parameters:

void *aWorkspace – USB Host driver workspace
void *aCallbackWorkspace – Workspace passed to callback.
const T_USBHostCallbacks *aCallbackAPI – Pointer to callback API routines.

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code.

5.16.17. SetConfiguration()

Description:

Instruct the USB device to switch to the given configuration number, usually 1.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aDeviceAddress – Which device to affect.
TUInt8 aConfigurationNum – Configuration to become (usually 1).

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code.

5.16.18. SetControlPacketSize()

Description:

Configures the maximum packet size of the control pipe to a specific device.

Parameters:

void *aWorkspace – USB Host driver workspace
TUInt8 aDeviceAddress – USB device address
TUInt16 aPacketSize – Maximum size packet (usually 64)

Results:

T_uezError – Results UEZ_ERROR_NONE, or error code.

6.0 Libraries

6.1 Light Weight IP (lwIP) TCP/IP Stack

lwIP is a small independent implementation of the TCP/IP protocol suite that has been developed by Adam Dunkels at the Computer and Networks Architectures (CNA) lab at the Swedish Institute of Computer Science (SICS).

The focus of the lwIP TCP/IP implementation is to reduce resource usage while still having a full scale TCP. This making lwIP suitable for use in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code ROM.

lwIP features:

- IP (Internet Protocol) including packet forwarding over multiple network interfaces
- ICMP (Internet Control Message Protocol) for network maintenance and debugging
- IGMP (Internet Group Management Protocol) for multicast traffic management
- UDP (User Datagram Protocol) including experimental UDP-lite extensions
- TCP (Transmission Control Protocol) with congestion control, RTT estimation and fast recovery/fast retransmit
- raw/native API for enhanced performance
- Optional Berkeley-like socket API
- DNS (Domain names resolver)
- SNMP (Simple Network Management Protocol)
- DHCP (Dynamic Host Configuration Protocol)
- AUTOIP (for IPv4, conform with RFC 3927)
- PPP (Point-to-Point Protocol)
- ARP (Address Resolution Protocol) for Ethernet

In the µEZ™ implementation, two interfaces have been provided. The BSD API and the Netconn API.

6.1.1. BSD API

6.1.1.1. accept()

Description:

This function will return a socket connected to the remote socket with the first connection request in the connection queue. The address structure of the connected socket is returned in the struct addr.

Parameters:

int s – socket to accept
struct sockaddr *addr – socket address connected
socklen_t *addrlen – size of addr.

Results:

int – Returns -1 on error, the newly accepted socket otherwise.

6.1.1.2. bind()

Description:

Binds the socket s to the port specified in the address structure name.

Parameters:

int s – socket to bind
struct sockaddr *name – socket address connected
int namelen – length of name

Results:

int – Returns 0 on success, -1 on error.

6.1.1.3. close() / closesocket()

Description:

Close a socket.

Parameters:

int s – socket to close

Results:

int – Returns 0 on success, -1 on error.

6.1.1.4. getpeername()

Description:

The getpeername() function is used to get the name of a connected peer. The name was assigned when a connect() or accept() function was successfully completed.

Parameters:

int s – socket to get name
struct sockaddr *name – structure to hold 'name'
int *namelen – Pointer to amount of space available. on return it contains the actual size of the name returned (in bytes).

Results:

int – Returns 0 if successful, else -1.

6.1.1.5. getsockname()

Description:

Get socket name assigned to a socket, which is the address of the local endpoint and was assigned with a bind() function.

Parameters:

int s – socket to get name
struct sockaddr *name – structure to hold 'name'
int *namelen – Pointer to amount of space available. on return it contains the actual size of the name returned (in bytes).

Results:

int – Returns 0 if successful.

6.1.1.6. getsockopt()

Description:

Returns an option setting associated with a socket.

Parameters:

int s – socket to manipulate
int level -- SOL_SOCKET, IPPROTO_IP, IPPROTO_TCP, or IPPROTO_UDPLITE
int optname – specific option for level
char *optval – Value of option. Most options take a int value.
int optlen – length of value (in bytes)

Results:

int – Returns 0 if successful.

6.1.1.7. ioctl()

Description:

The ioctl() function manipulates I/O controls associated with a socket. In particular, the sensing of out-of-band data marks and the enabling of non-blocking I/O can be controlled.

Parameters:

int s – socket to control
unsigned long request – request type
char *argp – pointer to argument for request

Results:

int – Returns 0 if successful, -1 if an error.

6.1.1.8. listen()

Description:

The listen() function indicates the application program is ready to accept connection requests arriving at a socket of type SOCK_STREAM. The connection request is queued (if possible) until accepted with an accept() function.

Parameters:

int s – socket to listen on

int backlog – defines the maximum number of pending connections that may be queued.

Results:

int – Returns 0 if successful, -1 if an error.

6.1.1.9. recv()

Description:

The recv() function is used to receive incoming data that has been queued for a socket. This function is normally used to receive a reliable, ordered stream of data bytes on a socket of type SOCK_STREAM.

Parameters:

int s – socket to receive data

char *buf – pointer to buffer to receive bytes.

int len – number of bytes allowed in buffer

int flags – set to MSG_OBB to receive out of band data, otherwise leave 0.

Results:

int – Returns the number of bytes received. A value of 0 is returned if the end-of-file (socket closed). A -1 return value indicates an error.

6.1.1.10. recvfrom()

Description:

The recvfrom() function is used to receive incoming data that has been queued for a socket. This function normally is used to receive datagrams on a socket of type SOCK_DGRAM.

Parameters:

int s – socket to receive data

char *buf – pointer to buffer to receive bytes.

int len – number of bytes allowed in buffer

int flags – set to MSG_OBB to receive out of band data, otherwise leave 0.

struct sockaddr *from – address of sender

int *fromlen – length of the datagram

Results:

int – Returns the number of bytes received. A value of 0 is returned if the end-of-file (socket closed). A -1 return value indicates an error.

6.1.1.11. select()

Description:

The select() function is used to synchronize the processing of several sockets operating in non-blocking mode. Sockets that are ready for reading, ready for writing, or have a pending exceptional condition can be selected. If no sockets are ready for processing, the select() function can block indefinitely or wait for a specified period of time (which may be zero) and then return.

Parameters:

int nfds – number of socket descriptors
fd_set *readfds – descriptors waiting for read
fd_set *writefds – descriptors waiting for write
fd_set *exceptfds – descriptors waiting for exceptions
struct timeval *timeout – timeout value

Results:

int – Returns the number of ready descriptors in the descriptor sets, or 0 if the time limit expired. A value of -1 is returned on an error.

6.1.1.12. send()

Description:

The send() function is used to send outgoing data on a connected socket s (usually a SOCK_STREAM).

Parameters:

int s – socket to send data
char *msg – pointer to buffer to send bytes.
int len – number of bytes to send
int flags – set to MSG_OBB to receive out of band data, otherwise leave 0.

Results:

int – Returns the number of bytes sent. Otherwise, a -1 return value indicates an error.

6.1.1.13. sendto()

Description:

The sendto() function is used to send outgoing data on a connected or unconnected socket (normally of type SOCK_DGRAM).

Parameters:

int s – socket to send data
char *msg – pointer to buffer to send bytes.
int len – number of bytes to send
int flags – set to MSG_OBB to receive out of band data, otherwise leave 0.
struct sockaddr *to – address to send to
int tolen – total length to send

Results:

int – Returns the number of bytes sent. Otherwise, a -1 return value indicates an error.

6.1.1.14. setsockopt()

Description:

Manipulates options associated with a socket.

Parameters:

int s – socket to manipulate
int level -- SOL_SOCKET, IPPROTO_IP, IPPROTO_TCP, or IPPROTO_UDPLITE
int optname – specific option for level
char *optval – Value of option. Most options take a int value.
int optlen – length of value (in bytes)

Results:

int – Returns 0 if successful.

6.1.1.15. shutdown()

Description:

Close one end of a full-duplex connection. Currently, the full connection is closed.

Parameters:

int s – socket to shutdown
int how – currently not used

Results:

int – Returns 0 on success, -1 on error.

6.1.1.16. socket()

Description:

Create an endpoint for communication.

Parameters:

int domain – AF_INET (not used in this implementation)
int type – SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW
int protocol – specific protocol, typically 0 for default.

Results:

int – Returns the socket (positive number) on success. Otherwise, a -1 return value indicates an error.

6.1.2. Netconn API

6.1.2.1. netconn_accept()

Description:

Accept a new connection on a TCP listening netconn.

Parameters:

struct netconn *conn – the TCP listening netconn

Results:

struct netconn * -- the newly accepted netconn, or NULL on timeout

6.1.2.2. netconn_bind()

Description:

Bind a netconn to a specific local IP address and port. Binding one netconn twice might not always be checked correctly.

Parameters:

struct netconn *conn – netconn to bind

struct ip_addr *addr – the local IP address to bind the netconn to (use IP_ADDR_ANY to bind to all addresses)

u16_t port – the local port to bind the netcoon to (not used for RAW).

Results:

err_t -- ERR_OK if the connection was deleted

6.1.2.3. netconn_close()

Description:

close a TCP netconn (doesn't delete it)

Parameters:

struct netconn *conn – the TCP netconn to close

Results:

err_t -- ERR_OK if netconn was closed, any other err_t on error.

6.1.2.4. netconn_connect()

Description:

Connect a netconn to a specific remote IP address and port

Parameters:

struct netconn *conn – the netconn to connect

struct ip_addr *addr – the remote IP address to connect to

u16_t port – the remote port to connect to (not used for RAW)

Results:

err_t -- ERR_OK if connected

6.1.2.5. **netconn_delete()**

Description:

Close a netconn 'connection' and free its resources. UDP and RAW connections are completely closed, TCP connections may still be in a waitstate after this returns.

Parameters:

struct netconn *conn – netconn to be deleted

Results:

err_t -- ERR_OK if the connection was deleted

6.1.2.6. **netconn_disconnect()**

Description:

Disconnect a netconn from its current peer (only valid for UDP netconns).

Parameters:

struct netconn *conn – the netconn to disconnect

Results:

err_t -- ERR_OK if disconnected

6.1.2.7. **netconn_getaddr()**

Description:

Get the local or remote IP address and port of a netconn. For RAW netconns, this returns the protocol instead of a port.

Parameters:

struct netconn *conn – netconn to query
struct ip_addr *addr – a pointer to the save the IP address
u16_t *port – a pointer to save the port (or protocol for RAW)
u8_t local – 1 to get the local IP address, 0 to get the remote address.

Results:

err_t – ERR_CONN for invalid connections, ERR_OK if the information was retrieved.

6.1.2.8. **netconn_gethostbyname()**

Description:

Execute a DNS query, only one IP address is returned.

Parameters:

const char *name – a string representation of the DNS host name to query.
struct ip_addr *addr – a preallocated struct ip_addr where to store the resolved IP address.

Results:

err_t – ERR_OK if resolved, ERR_MEM if memory error, ERR_ARG if dns client not initialized or invalid hostname, ERR_VAL if dns server response was invalid.

6.1.2.9. netconn_listen_with_backlog()

Description:

Set a TCP netconn into listen mode. Note: netconn_listen() calls netconn_listen_with_backlog() with a default backlog of 255.

Parameters:

struct netconn *conn – the netconn to listen
u8_t backlog – the listen backlog

Results:

err_t -- ERR_OK if netconn was set to listen

6.1.2.10. netconn_recv()

Description:

Receive data (in form of a netbuf containing a packet buffer) from a netconn.

Parameters:

struct netconn *conn – the netconn to from which to receive data

Results:

struct netbuf * -- a new netbuf containing data or NULL on memory error or a timeout.

6.1.2.11. netconn_send()

Description:

Send data over a UDP or RAW netconn (that is already connected)

Parameters:

struct netconn *conn – the UDP or RAW netconn over which to send data
struct netbuf *buf – a netbuf containing the data to send

Results:

err_t – ERR_OK if data was sent, any other err_t on error.

6.1.2.12. netconn_sendto()

Description:

Send data (in form of a netbuf) to a specific remote IP address and port. Only to be used for UDP and RAW netconns (not TCP).

Parameters:

struct netconn *conn – the netconn over which to send data
struct netbuf *buf – a netbuf containing the data to send
struct ip_addr *addr – the remote IP address to which to send the data
u16_t port – the remote port to which to send the data

Results:

err_t – ERR_OK if data was sent, any other err_t on error.

6.1.2.13. netconn_type()

Description:

Get the type of a netconn.

Parameters:

struct netconn *conn – netconn to get the type

Results:

netconn_type – type of the netconn

6.1.2.14. netconn_write()

Description:

Send data over a TCP netconn.

Parameters:

struct netconn *conn – the TCP netconn over which to send data

const void *dataptr – pointer to the application buffer that contains data to send.

int size – Size of the application data to send.

u8_t apiflags – combination of flags. NETCONN_COPY (0x01) means data will be copied into memory belonging to the stack and NETCONN_MORE (0x02) for TCP connection, PSH flag will be set on last segment sent.

Results:

err_t – ERR_OK if data was sent, any other err_t on error.

6.2 Simple Windowing Interface Manger (SWIM)

The SWIM library is a small window manager library created by NXP for embedded applications that need a light weight graphics library for drawing pixels, lines, boxes, text, and images on LCD devices.

6.2.1. Concepts

6.2.2. Window Routines

6.2.2.1. swim_window_open()

Description:

Initializes a window and the default values for the window. Then clears and draws the window. Fonts are defaulted as not transparent.

Parameters:

SWIM_WINDOW_T *win – Preallocated windows structure to fill
INT_32 xsize – Physical horizontal dimension of the display
INT_32 ysize – Physical vertical dimension of the display
COLOR_T *fbaddr – Address of the display's frame buffer
INT_32 xwin_min – Physical window left coordinate
INT_32 ywin_min – Physical window top coordinate
INT_32 xwin_max – Physical window right coordinate
INT_32 ywin_max – Physical window bottom coordinate
INT_32 border_width – Width of the window border in pixels
COLOR_T pcolor – Pen color (border color)
COLOR_T bkcolor – Background color (fill of background)
COLOR_T fcolor – Fill color (future fill commands)

Results:

BOOL_32 – TRUE if window was initialized correctly, otherwise FALSE

6.2.2.2. swim_window_open_noclear()

Description:

Initializes a window and the default values for the window. Window is not cleared and drawn. Fonts are defaulted as transparent.

Parameters:

SWIM_WINDOW_T *win – Preallocated windows structure to fill
INT_32 xsize – Physical horizontal dimension of the display
INT_32 ysize – Physical vertical dimension of the display
COLOR_T *fbaddr – Address of the display's frame buffer
INT_32 xwin_min – Physical window left coordinate
INT_32 ywin_min – Physical window top coordinate
INT_32 xwin_max – Physical window right coordinate
INT_32 ywin_max – Physical window bottom coordinate
INT_32 border_width – Width of the window border in pixels
COLOR_T pcolor – Pen color (border color)
COLOR_T bkcolor – Background color (fill of background)
COLOR_T fcolor – Fill color (future fill commands)

Results:

BOOL_32 – TRUE if window was initialized correctly, otherwise FALSE

6.2.2.3. swim_window_close()

Description:

Releases a window for future use.

Parameters:

SWIM_WINDOW_T *win – Window identifier

Results:

Nothing

6.2.2.4. swim_get_horizontal_size()

Description:

Get the virtual window horizontal size

Parameters:

SWIM_WINDOW_T *win – Window identifier

Results:

INT_32 – The virtual window horizontal size.

6.2.2.5. swim_get_vertical_size()

Description:

Get the virtual window vertical size

Parameters:

SWIM_WINDOW_T *win – Window identifier

Results:

INT_32 – The virtual window vertical size.

6.2.3. Drawing Routines

6.2.3.1. swim_set_pen_color()

Description:

Sets the pen color.

Parameters:

SWIM_WINDOW_T *win – Window identifier
COLOR_T pen_color – New pen color

Results:

Nothing

6.2.3.2. swim_set_fill_color()

Description:

Sets the fill color.

Parameters:

SWIM_WINDOW_T *win – Window identifier
COLOR_T pen_color – New fill color

Results:

Nothing

6.2.3.3. swim_set_bkg_color()

Description:

Sets the color used for backgrounds.

Parameters:

SWIM_WINDOW_T *win – Window identifier
COLOR_T bkg_color – New background color

Results:

Nothing

6.2.3.4. swim_put_pixel()

Description:

Puts a pixel at (x, y) in the pen color.

Parameters:

SWIM_WINDOW_T *win – Window identifier
INT_32 x1 – x position
INT_32 y1 – y position

Results:

Nothing

6.2.3.5. swim_put_line()

Description:

Draw a line in the virtual window with clipping between (x1, y1) and (x2, y2).

Parameters:

SWIM_WINDOW_T *win – Window identifier
INT_32 x1 – x position of line start
INT_32 y1 – y position of line start
INT_32 x2 – x position of line end
INT_32 y2 – y position of line end

Results:

Nothing

6.2.3.6. swim_put_box()

Description:

Place a box with corners (x1, y1) and (x2, y2). The border is 1 pixel wide and in the pen color. The middle of the box is filled with the background fill color.

Parameters:

SWIM_WINDOW_T *win – Window identifier
INT_32 x1 – Virtual left position of box
INT_32 y1 – Virtual upper position of box
INT_32 x2 – Virtual right position of box
INT_32 y2 – Virtual bottom position of box

Results:

Nothing

6.2.3.7. swim_clear_screen()

Description:

Fills the draw area of the window with the selected color.

Parameters:

SWIM_WINDOW_T *win – Window identifier
COLOR_T color – Color to place in the window

Results:

Nothing

6.2.3.8. swim_put_diamond()

Description:

Draw a diamond in the virtual window.

Parameters:

SWIM_WINDOW_T *win – Window identifier
INT_32 x – Virtual X position of the diamond
INT_32 y – Virtual Y position of the diamond
INT_32 rx – Radius for horizontal
INT_32 ry – Radius for vertical

Results:

Nothing

6.2.4. Font Routines

6.2.4.1. swim_set_font()

Description:

Select the active font.

Parameters:

SWIM_WINDOW_T *win – Window identifier
const FONT_T *font – Pointer to the selected font data structure

Results:

Nothing

6.2.4.2. swim_get_font_height()

Description:

Returns the active font's height in pixels.

Parameters:

SWIM_WINDOW_T *win – Window identifier

Results:

INT_16 – The height of the active font in pixels

6.2.4.3. swim_set_font_trasparency()

Description:

Enables and disables font backgrounds. When set, the font background will not be drawn in the background color (useful for painting text over pictures).

Parameters:

SWIM_WINDOW_T *win – Window identifier
INT_32 trans – When not 0, the font backgrounds will NOT be drawn.

Results:

Nothing

6.2.4.4. swim_get_xy()

Description:

Returns the X, Y pixel coordinates for the next text operation.

Parameters:

SWIM_WINDOW_T *win – Window identifier
INT_32 *x – Pointer to returned virtual X position of start of text
INT_32 *y – Pointer to returned virtual Y position of start of text

Results:

Nothing

6.2.4.5. swim_set_xy()

Description:

Sets the X, Y pixel coordinates for the next text operation.

Parameters:

SWIM_WINDOW_T *win – Window identifier
INT_32 x – Virtual X position of start of text
INT_32 y – Virtual Y position of start of text

Results:

Nothing

6.2.4.6. swim_put_char()

Description:

Puts a character in the window at the current text (x, y) location.

Parameters:

SWIM_WINDOW_T *win – Window identifier
INT_32 x – Virtual X position of start of text
INT_32 y – Virtual Y position of start of text

Results:

Nothing

6.2.4.7. swim_put_newline()

Description:

Set the text pointer for the next text character operation to the beginning of the following line. If the following line exceeds the window size, perform a line scroll.

Parameters:

SWIM_WINDOW_T *win – Window identifier

Results:

Nothing

6.2.4.8. swim_put_text()

Description:

Puts a string of text in a window at the current location. For newline characters, a newline will occur instead of a character output.

Parameters:

SWIM_WINDOW_T *win – Window identifier
const CHAR *text – Text string to output in window

Results:

Nothing

6.2.4.9. swim_put_ltext()

Description:

Puts a string of text in a window, but will adjust the position of a word if the word length exceeds the edge of the display. For newline characters, a newline will occur instead of a character output.

Parameters:

SWIM_WINDOW_T *win – Window identifier
const CHAR *text – Text string to output in window

Results:

Nothing

6.2.4.10. swim_put_text_xy()

Description:

Put text at x, y (char) position on screen.

Parameters:

SWIM_WINDOW_T *win – Window identifier
const CHAR *text – Text string to output in window

Results:

Nothing

6.3 USB Device Generic Bulk Library

The Generic Bulk library is an example library that shows how to enable the target to become a USB Device that uses the 'usblib' PC and Linux library to communicate over USB bulk endpoints.

6.3.1. Concepts and Overview

The Generic Bulk library was created with the idea that a developer may wish to communicate between the target (device) and the PC (host) similar to a serial port. Data is streamed from the host to the target and back. To accomplish this, the uEZ GenericBulk library uses a GenericBulkGet and GenericBulkPut command with queues to block until data is ready or sent. While the get and put routines are blocking, the GenericBulkMonitor is monitoring USB communications and doing the work of taking data off the send queue, sending it out, and then stuffing responses back on the receive queue. This is all done by making continuous calls to ProcessEndpoints on the USB Device Driver.

The routines GenericBulkBulkIn() and GenericBulkBulkOut() handle the data passed between the USB Device Driver and the queues. They read one block at a time and pass it on.

Bulk endpoints are used to send and receive data. Because bulk data is always 64 bytes (or larger), the data that is sent and received uses a one byte header that declares the actual length of the data used in the block. In this way, smaller blocks of data can be passed back and forth.

6.3.1.1. On the PC Side

Included with the source code is an example Visual C++ 6.0 project called Utilities\USB\libusb_bulk_example\libusb_example. This example starts up, sends "Hello" to the device, and then monitors all data received to outputs the data to the console screen.

6.3.2. Usage

To use the GenericBulk library, just call GenericBulkInitialize() providing a structure to any callbacks you may want (in specific, if you want a callback routine each time the output queue becomes empty). The hardware is initialized and monitor task is created to handle data transfers in and out of the queues. Use GenericBulkPut and GenericBulkGet to send and receive data.

6.3.3. Library Routines

6.3.3.1. GenericBulkInitialize()

Description:

Initialize the GenericBulk system. The hardware is initialized and the queues created. A monitoring task called "USBDevice" is created. Once completed, the target can be plugged into a USB Host port.

NOTE: If using both a USB Device and USB Host configuration, call GenericBulkInitialize() after the USB Host has been initialized.

Parameters:

T_GenBulkCallbacks *aCallbacks – Pointer to callback routines. Currently only a single callback when the output queue (from device to host) becomes empty.

Results:

T_uezError – UEZ_ERROR_NONE if successful, else error code.

6.3.3.2. GenericBulkPut()

Description:

Put character into the output queue (from device to host) or timeout trying.

Parameters:

TUInt8 c – Character to output

TUInt32 aTimeout – Length of time in milliseconds to wait before timing out.

Results:

TBool – Returns ETrue if successful, else EFalse.

6.3.3.3. GenericBulkGet()

Description:

Retrieve a character from the incoming queue (from host to device).

Parameters:

TUInt32 aTimeout – Length of time in milliseconds to wait before timing out.

Results:

TInt32 – Character received (0 – 0xFF) or -1 if failed.

6.3.4. More Information about libusb library

The Windows version of the libusb library is available at http://libusb-win32.sourceforge.net/

The Linux version of the libusb library is available at http://libusb.wiki.sourceforge.net/

Check both sites for up to date versions.

7.0 The uEZ™ Boot Process

The uEZ™ system goes through following steps before main() is called in the application code. The following steps assumes that any separate bootloader has already activated uEZ™.

1) Execution starts in the .s or bootstrap code where the memory is initialized and the stacks are configured.
2) UEZBSPInit() is called (see file Source/BSP/uEZBSPInit.c)
    a. UEZBSP_CPU_PinConfigInit is called to initialize the CPU pin muxing and GPIO pins.
    b. Interrupts are then reset (and kept off).
    c. Low level BSP serial port is initialized via UEZBSPSerialInit(). If serial debug is being used, the routines should work with interrupts off.
    d. Memory management is then started with a call to UEZBSPSRAMInit() and UEZBSPSDRAMInit().
    e. UEZBSPHeapInit() is called to initialize any custom memory managers that may have not already been initialized by the bootstrap code.
    f. UEZSystemInit() is an internal call to uEZ™ to setup the registries and the core uEZ™ features.
    g. uEZProcessorServicesInit() then sets up all the hardware HAL device drivers provided by the processor.
    h. Next, uEZPlatformInit() is called to setup all the device drivers (typically non-HAL based) provided by all devices external to the processor.
    i. The uEZPlatformStartup task is now created and the RTOS started (via UEZSystemMainLoop()). uEZPlatformStartup executes any RTOS dependent initialization.
    j. The uEZPlatformStartup task then creates the Main task with an entry point at main().
    k. Routine main() is now runs. NOTE: This routine never returns.
3) main() is now executing.

8.0 Setting up a New Processor

8.1 Overview

uEZ™ was designed to allow portability between multiple processors with little to no changes to the code. Porting uEZ™ to new processors requires the addition of two primary steps: installing a compatible RTOS and adding support files for the processor.

8.2 Processor Specific RTOS

A RTOS with the new processor feature must be added first. If the RTOS does not support the new processor, then the RTOS will have to be modified first. Follow the instructions for porting the RTOS as provided by the maker of the RTOS. Place the resulting build in the Source/RTOS directory. If the code is to be submitted back to the sourceforge.net, the RTOS should be modified by adding the new processor supported, not just replacement.

8.3 Adding uEZ™ Processor Device Driver Files

In effect, adding a new processor is simply the addition of a collection of HAL device drivers. All processor device drivers are put in one directory under the manufacturer and processor name as follows:

/Source/Processor/<manufacturer>/<processor name>

For example,

/Source/Processor/NXP/LPC2478

specifies the directory for all the LPC2478 files.

Each processor specific HAL driver should be prefixed with the processor name with an underbar (e.g. LPC2478_) to avoid filename conflicts.

All processor types are defined in the file Include/Config.h. Add the new processor to the list of processor types in MANUFACTURER_PROCESSOR format and in all caps (e.g., NXP_LPC2478) and use the next sequential number. Change the #define UEZ_PROCESSOR to be the newly defined MANUFACTURER_PROCESSOR name.

To ensure that the correct device drivers compile, in each processor HAL device driver .c and/or .s file include the following header and footer:
Header:

```
#include "Config.h"
#if (UEZ_PROCESSOR==MANUFACTURER_PROCESSOR)
```

Footer:

```
#endif // (UEZ_PROCESSOR==MANUFACTURER_PROCESSOR)
```

Finally, add all these new files to the project make file. For example, if using Rowley Crosswork's CrossStudio, to the Project Explore, create a new folder under processors with a matching name, right click on the new folder, and select "Add Existing File…". Select all the files needed by the processor and click **Open**.

8.4 Processor Initialization

Create a final file called <Processor>_HALInit.c (e.g., LPC2478_HALInit.c) copying an existing one from another project if necessary and add it the project. This file will also need the headers and footers as previously described.

This file only contains one routine called uEZProcessorServicesInit() and is called early in the boot sequence to register and configure all the hardware devices specific to this processor.

In general, each peripheral follows these two basic steps.

8.4.1. Step 1: Register the HAL Driver Interface

Call HALInterfaceRegister() with the name and HAL interface. If the hardware device is needed by another HAL device, the return pointer to the workspace can be optionally returned. For example:

```
HALInterfaceRegister(
        "GPIO0",
        (T_halInterface *)&GPIO_LPC2478_Port0_Interface,
        0,
        &p_port);
```

In this case, the HAL driver is defined by GPIO_LPC2478_Port0_Interface, will be defined in the registry as "GPIO0", and returns a pointer to the T_halWorkspace in p_port.

The HAL driver interfaces are defined inside each of the individual HAL drivers. For each interface, add the following line at the top:

```
extern const HAL_GPIOPort GPIO_LPC2478_Port0_Interface;
```

In this case, a HAL_GPIOPort is defined, but the type will change with the different class types.

8.4.2. Step 2: Configure the HAL Driver

Next, call an appropriate configure routine, if necessary. Each device driver will have a different routine for configuring, and some will have none. HAL drivers may depend on each other and the returned workspace pointer will be passed in to the next configure routine. In this way, they can be linked together. When configuration is required, a specific routine know only by that driver and with the word Configure in it will be used. To continue the GPIO0 port initialization example, the following is used.

```
((HAL_GPIOPort *)(p_port->iInterface))->Configure(
    p_port,
    0xFFFFFFFF);
```

In this case, the GPIO driver wants to know which of the pins to allow in the port definition and a value of 0xFFFFFFFF signifies all 32 port pins are available on this processor version.

8.4.3. Miscellaneous Initialization

Any additional configuration needed may also go in the uEZProcessorServicesInit() routine, but it is recommended that this be kept to a minimum. The uEZProcessorServicesInit()'s primary purpose is to register devices, not go through specific processor initialization. These steps should be handled by the bootstrap code.

8.5 Interrupt Table

A core feature of all processors is the interrupt table. Interrupts are referenced by 'channels' and priorities. Although possibly already provided by the RTOS, the following routines must be provided to be compatible with the uEZ system (see Include/HAL/Interrupt.h for complete function prototypes).

```
InterruptsReset
InterruptRegister
InterruptUnregister
InterruptEnable
InterruptDisable
InterruptDisableAllRegistered
InterruptEnableAllRegistered
```

8.6 Testing a New Processor

It is recommended that a new processor be tested with a known good hardware configuration (bootstrap works, memory initialized, and ability to program code into part) along with a good debugger to step through the code. Use the debugger to step through the uEZProcessorServicesInit() and make sure the processor is setup in the proper order. Unfortunately, at this point, each device must be tested one by one with an appropriate test program and is target dependent. Usually it is best to test one hardware component at a time, possibly leaving out the others in uEZProcessorServicesInit() until confirmed.

9.0 Setting up a New Platform

9.1 Overview

A uEZ™ platform is viewed as a complete hardware solution using a specific processor and a collection of external peripherals. Device drivers provide the glue between the uEZ™ system, application, RTOS, and low level HAL drivers. Therefore, the primary role of the platform is defining of all available external hardware and how they connect.

9.2 Platform Files

All files for the platform are stored in the following directory path.

    /Source/Platform/<manufacturer>/<platform name>

The platform file is usually broken into two files: BSP and Platform in the following format:

    /Source/Platform/<manufacturer>/<platform name>/<platform>_uEZBSP.c
    /Source/Platform/<manufacturer>/<platform name>/<platform>_uEZPlatform.c

The <platform>_uEZBSP.c file contains the following functions (if enabled in Config.h):

    UEZBSP_CPU_PinConfigInit()
    UEZBSPSDRAMInit()
    UEZBSPSerialInit()
    UEZBSPSRAMInit()
    UEZBSPHeapInit()

The <platform>_uEZPlatform.c file contains the following required function:

    uEZPlatformInit()

Unlike the processor directory where all HAL drivers are stored in this path, the platform stores all device drivers in the /Source/Device subdirectories.

9.3 Platform Configuration

All platform types are defined in the file Include/Config.h. Add the new platform to the list of platform types in MANUFACTURER_PLATFORM format and in all caps (e.g., FDI_CARRIER) and use the next sequential number. Change the #define UEZ_PLATFORM to be the newly defined MANUFACTURER_PLATFORM name.

To ensure that the correct files compile, in each platform .c file include the following header and footer:

Header:

```
#include "Config.h"
#if (UEZ_PROCESSOR==MANUFACTURER_PLATFORM)
```

Footer:

```
#endif // (UEZ_PROCESSOR==MANUFACTURER_PLATFORM)
```

9.4 Platform Initialization

In file >/<platform>_uEZPlatform.c, uEZPlatformInit () and is called in the boot sequence to register and configure all the hardware devices specific to this platform. At this point, the processor has already registered all of its devices and can be found using HALInterfaceFind. Most, if not all, of the device drivers will link to HAL devices.

At this point, the RTOS has been initialized, but is not running. The code must not use RTOS features other than to create tasks, semaphores, queues, etc. NOTE: Tasks will run when the RTOS is started.

9.4.1. Device Driver Registration and Initialization

Each device peripheral is initialized using the following steps.

9.4.1.1. Step 1: Register the Device Driver Interface

Call UEZDeviceTableRegister () with the name and device interface. If the hardware device is needed by another device, the return pointer to the workspace can be optionally returned. For example:

```
UEZDeviceTableRegister(
        "I2C0",
        (T_uezDeviceInterface *)&I2CBus_Generic_Interface,
        0,
        &p_i2c0);
```

In this case, the device driver is defined by I2CBus_Generic_Interface, will be defined in the device registry as “I2C0”, and returns a pointer to the T_uezDeviceWorkspace in p_i2c0.

The device driver interfaces are defined inside each of the individual device drivers. For each interface, add the following line at the top:

```
extern const DEVICE_I2C_BUS I2CBus_Generic_Interface;
```

In this case, a DEVICE_I2C_BUS class type is defined for the interface. Each device will have a different interface type.

9.4.1.2. Step 2: Configure the Device Driver

Next, call an appropriate configure routine, if necessary. Each device driver will have a different routine for configuring, although some will have none. Device drivers usually depend on HAL drivers as well as each other. The returned workspace pointer can be passed into other configure routines linking together the drivers.

When configuration is required, a specific routine defined by that driver (with the word Configure in it) is used. To continue the I2C Bus initialization example, the following is used.

```
I2C_Generic_Configure(p_i2c0, "I2C0");
```

In this case, the I2C Bus driver wants the HAL driver registry name and will connect to it.

9.4.2. Additional Startup Steps

Additional functions can be called to start any libraries or tasks that are needed for running the platform. Again, at this point the RTOS is not running, so tasks can only be created but not executed. For example, the TCP/IP stack can configure and prepare its memory and root task, but will not be running.

9.4.3. Platform Startup Task

Once uEZPlatformInit() is completed, the task uEZPlatformStartup is created and the RTOS is started. uEZPlatformStartup is the second phase of initialization where the platform chooses to do any RTOS dependent initialization steps. It should finally end by creating the task called "Main" and starting main(). For example,

```
    UEZTaskCreate(
        (T_uezTaskFunction)main,
        "Main",
        512,
        0,
        UEZ_PRIORITY_NORMAL,
        &G_mainTask);
```

10.0 Appendix: Example Widget Driver Implementation

This section details how to create a new set of drivers to interface to a new hardware device called a Widget. For this example, let's assume that the Widget is a Widgetness device to turns on or off a light. Widgetness devices have two commands: TurnOn and TurnOff.

Most implementations have three layers: Library, Device Driver, and HAL Driver. In this example, we'll work from the bottom up.

10.1 HAL Driver

At the bottom most level, the HAL driver provides direct access to the hardware and assumes that it has exclusive control of the hardware on each function call. Because there may be multiple types of Widgets, each HAL Driver uses a HAL Driver Type structure which is defined in the Include/HAL directory. For this example, we'll start by making the file Include/HAL/Widget.h as follows:

```
// File: Include/HAL/Widget.h

#ifndef _WIDGET_H_
#define _WIDGET_H_
#include <uEZTypes.h>
#include <HAL/HAL.h>

typedef struct {
    T_halInterface iInterface;

    T_uezError (*TurnOn)(void *aWorkspace);
    T_uezError (*TurnOff)(void *aWorkspace);
} HAL_Widget;

#endif // _WIDGET_H_
```

The above file defines HAL_Widget. All Widget devices must use this interface style for future compatibility. It declares that widgets can be turned on or off and nothing else. The T_halInterface iInterface structure is a standard header on all HAL interfaces and will be explained below.

The HAL driver itself is defined in the /Source/Devices/{DeviceType}/{Company}/{SpecificDevice} directory where {DeviceType} is Widget in this case, {Company} is the maker of the device, and {SpecificDevice} is the particular model of widget. Assuming the company is Acme and we're working on the W1000, it would go to the directory "/Source/Devices/Widget/Acme/W1000".

File names will start width the model number followed usually by "main". In this case, we create the file "widget1000_main.c". Routines will also use a "{DeviceType}_{Company}_{SpecificDevice}_{function}" name to avoid name conflicts and to aide debugging. For example, our widget HAL driver may appear as follows:

```
// File: widget1000_main.c

// Workspace definition
typedef struct {
    const HAL_Widget *iDevice;
    // private data goes here
} T_Widget_Acme_W1000_Workspace;

// Functions:
T_uezError Widget_Acme_W1000_TurnOn(void *aWorkspace)
{
    T_Widget_Acme_W1000_Workspace *p =
(T_Widget_Acme_W1000_Workspace *)aWorkspace;
    T_uezError error = UEZ_ERROR_NONE;

    // Driver specific code goes here using variable p
    return error;
}

T_uezError Widget_Acme_W1000_TurnOff(void *aWorkspace)
{
    T_Widget_Acme_W1000_Workspace *p =
(T_Widget_Acme_W1000_Workspace *)aWorkspace;
    T_uezError error = UEZ_ERROR_NONE;

    // Driver specific code goes here using variable p
    return error;
}

T_uezError Widget_Acme_W1000_InitializeWorkspace(void *aWorkspace)
{
    T_Widget_Acme_W1000_Workspace *p =
(T_Widget_Acme_W1000_Workspace *)aWorkspace;
    T_uezError error = UEZ_ERROR_NONE;

    // Set initial values in workspace
    return error;
}

const HAL_Widget Widget_Acme_W1000_Widget0 = {
    "Widget:Acme:W1000:Widget0",
    0x0100,
    Widget_Acme_W1000_InitializeWorkspace,
    sizeof(T_Widget_Acme_W1000_Workspace),
};
```

There are four main sections to the HAL driver. First, there is the workspace. This represents a private work space for this instance of the driver. For example, there may be 4 widgets on this hardware platform that use the same HAL interface. Each one has its own workspace. Second, there are the functions that operation on the workspace. Each function represents one unit of work that must be completed before another function (typically) is called. Third, is the routine xxx_InitializeWorkspace which is called once to setup the workspace. Finally, the HAL interface is defined and links up the above 3 sections. It provides a unique name for the interface, its version number (for compatibility tracking), the initialization routine, and the size of the workspace required.

When the BSP starts, it will register all HAL driver interfaces to the system and instantiate all the workspaces. It will do this by calling HALInterfaceRegister and pass in the interface structure. In this case, it is Widget_Acme_W1000_Widget0.

After registration, other software can find it by calling HALInterfaceFind and getting a pointer to the workspace. All workspaces start with a pointer to the interface to provide easy access to pointers to the interface routines. For example, if you have a T_halWorkspace * pointer to a Widget0 device, you would use the following syntax to turn the Widget on:

```
((T_Widget_Acme_W1000_Workspace *)p)->TurnOn(p);
```

Widget0's TurnOn function is called and it is given the private workspace.

For this example, let's assume that Widgets are only turned on for short periods of time and multiple tasks may be running and accessing the same Widget. In this case, the RTOS will need to semaphore the accesses. Since the HAL layer is built without knowing about a RTOS, a device driver is needed.

10.2 Device Driver

Device Drivers follow almost exactly the same rules as the HAL drivers. Each has its own workspace, interface, and registration. The main difference between the device driver and the HAL driver is how it interfaces with the RTOS.

For our widget example, we now define a system of 'widgets' that apply to all widgets that use the RTOS to control task blocking. Again, a new device driver type will be created, but this time it will be created in "Include/Device/Widget.h", as shown below:

```
// File: Include/Device/Widget.h
#include <uEZTypes.h>
#include <uEZDevice.h>

typedef struct {
    T_uezDeviceInterface iDevice;

    T_uezError (*Configure)(void *aWorkspace, HAL_Widget**
                  aWidget);
    T_uezError (*TurnOn)(void *aWorkspace);
    T_uezError (*TurnOff)(void *aWorkspace);
    T_uezError (*WaitTilOff)(void *aWorkspace);
} DEVICE_Widget;
```

Notice that we have a similar structure as HAL_Widget, but we've added WaitTilOff() and Configure() to the list. Configure() is used to link the Device Driver to a specific HAL Widget driver and workspace. WaitTilOff() demonstrates how more functionality can be added to at this level while being compatible with all hardware implementations.

The device driver may look like this:

```
// File: Source/Devices/Generic/Widget
#include <uEZ.h>
```

```
#include <Device/Widget.h>
#include <HAL/Widget.h>

typedef struct {
    const DEVICE_Widget iDevice;
    HAL_Widget **iWidget;
    T_uezSemaphore iSem;
} T_Widget_Generic_Workspace;

T_uezError Widget_Generic_InitializeWorkspace(void *aWorkspace)
{
    T_uezError error;
    T_Widget_Generic_Workspace *p = aWorkspace;

    // Setup semaphore for accesses
    error = UEZSemaphoreCreateBinary(&p->iSem);

    return error;
}

T_uezError Widget_Generic_TurnOn(void *aWorkspace)
{
    T_uezError error = UEZ_ERROR_NONE;
    T_Widget_Generic_Workspace *p = aWorkspace;

    // Attempt to grab the Widget's semaphore, or get a timeout
immediately
    error = UEZSemaphoreGrab(p->iSem, 0);

    // Have the hardware turn on
    (*p->iWidget)->TurnOn(p->iWidget);

    return error;
}

T_uezError Widget_Generic_TurnOff(void *aWorkspace)
{
    T_uezError error = UEZ_ERROR_NONE;
    T_Widget_Generic_Workspace *p = aWorkspace;
    DEVICE_Widget *pw = (DEVICE_Widget *)(p->iDevice);

    // Attempt to release the Widget's semaphore, or timeout
immediately
    error = UEZSemaphoreRelease(p->iSem, 0);

    // Turn off the unit
    (*p->iWidget)->TurnOff(p->iWidget);

    return error;
}

T_uezError Widget_Generic_WaitTilOff (void *aWorkspace)
{
    T_uezError error = UEZ_ERROR_NONE;
    T_Widget_Generic_Workspace *p = aWorkspace;

    // Attempt to grab the Widget's semaphore, implying it is now
off and able
    // to turn off.
    error = UEZSemaphoreGrab(p->iSem, UEZ_TIMEOUT_INFINITE);
    if (error)
        return error;

    // Release semaphore, we had it when it was off
    error = UEZSemaphoreRelease(p->iSem, 0);

    return error;
}

T_uezError Widget_Generic_Configure(void *aWorkspace, HAL_Widget
**aWidget)
{
    T_uezError error = UEZ_ERROR_NONE;
    T_Widget_Generic_Workspace *p = aWorkspace;

    p->iWidget = aWidget;

    return error;
}

const DEVICE_Widget Widget_Generic_Interface = {
    "Widget:Generic",
    0x0100,
    Widget_Generic_InitializeWorkspace,
    sizeof(T_Widget_Generic_Workspace),

    // Functions
    Widget_Generic_Configure,
    Widget_Generic_TurnOn,
    Widget_Generic_TurnOff,
    Widget_Generic_WaitTilOff
};
```

Like the HAL implementation, there are four main sections: the workspace definition, the workspace initialization, the routines that work on the workspace (and make calls to the RTOS and hardware), and the interface definition.

The iWidget field and Configure() routine needs to be explained. The iWidget field is a pointer to a pointer to the HAL_Widget interface. The first level pointer points to the workspace. Since all workspaces start with a pointer to their associated interfaces, this effectively is the same as a pointer to a pointer. By doing iWidget, the address is the workspace. By doing *iWidget, the resulting address is the workspace. In this way, a single pointer can be used to represent both two structures. When the following line is executed:

```
(*p->iWidget)->TurnOff(p->iWidget);
```

It means that the routine TurnOff is being called in *iWidget (the HAL's interface routines) and iWidget (the HAL Widget's workspace) is being passed to it.

The Configure() routine is called to link the device driver with a HAL driver. It can also be used to setup any other values in the private workspace as needed (e.g., default on/off state).

Finally, the platform BSP must call choose which devices it is using and register them in the device table with UEZDeviceTableRegister(). The interface is registered with a name and a unique handle is assigned to this instance. Device handles can be found by name using UEZDeviceTableFind. UEZDeviceTableGetWorkspace can be used to turn handles into workspaces. The BSP will also typically give the device a generic name so that the library and application layers can use the device without knowing any specific details about the hardware. For example, “SPI0” may be used for the first SPI bus in the system.

10.3 Library

The next layer is the library layer. Although applications can directly access the device drivers or HAL drivers, most users expect a simple to use interface. The library can serve as a simple wrapper around the device driver, or it can link the devices to other devices as needed. Below is an example widget library implementation:

```
// File: Library/uEZWidget.c
#include <Device/Widget.h>
T_uezError UEZWidgetOpen(char *aName, T_uezDevice *aDeviceHandle)
{
    T_uezError error;
    error = UEZDeviceTableFind(aName, aDeviceHandle);
    if (error)
        return error;
    // Turn widget on by default
    WidgetOn(*aDeviceHandle);
}

T_uezError UEZWidgetOn(T_uezDevice aWidget)
{
    T_uezError error;
    DEVICE_Widget **p;

    // Get workspace or return error
    error = UEZDeviceTableGetWorkspace(aWidget,
(T_uezDeviceWorkspace *)&p);
    if (error)
        return error;

     return (*p)->TurnOn((void *)p);
}

T_uezError UEZWidgetOff(T_uezDevice aWidget)
{
    T_uezError error;
    DEVICE_Widget **p;

    // Get workspace or return error
    error = UEZDeviceTableGetWorkspace(aWidget,
(T_uezDeviceWorkspace *)&p);
    if (error)
        return error;

     return (*p)->TurnOn((void *)p);
}

T_uezError UEZWidgetWaitTilOff(T_uezDevice aWidget)
{
    T_uezError error;
    DEVICE_Widget **p;

    // Get workspace or return error
    error = UEZDeviceTableGetWorkspace(aWidget,
(T_uezDeviceWorkspace *)&p);
    if (error)
```

```
            return error;

         return (*p)->WaitTilOff((void *)p);
    }

    T_uezError UEZWidgetClose(T_uezDevice aWidget)
    {
        // Turn widget off by default
        return WidgetOff(aDeviceHandle);
    }
```

10.4 Summary

In conclusion, the above examples show a full bottom to top view of a complete device system. Note that most devices will use pieces of existing systems. For example, if a new widget type is developed, either a new device driver or HAL driver can be built without changing the other two components. The top level application will not need to know about the specific hardware details. If the library is changed to improve the application's experience, the hardware drivers will not need to be changed. Each level abstracts and protects the other layers from change.