µEZ® v2.03 Software User's Guide

FDI Future Designs, Inc.
Your Development Partner
996 A Cleaner Way, Huntsville, AL 35805

For more information on FDI or our products please visit www.teamfdi.com.

NOTE: The inclusion of vendor software products in this kit does not imply an endorsement of the product by Future Designs, Inc.

FDI PN: MA00016
Revision: 2.03, 11/28/2012 10:59:00 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.
C. Stocker	5/13/11	Added Audio Codec, watchdog, DAC EINT, and I2S device drivers. Added uEZWAVFile, uEZWatchDog, uEZDAC, and uEZEINT. Added I2S HAL driver.
C. Stocker	7/12/11	Added uEZ Network API and uEZ INI Library.
L. Shields	7/14/2011	Additional text in uEZ Network API and uEZ INI Library.
L. Shields	9/1/2011	Added uEZHID routines.
L. Shields	10/18/2011	Added SetMatchCallback to HAL Timer and HAL PWM. Adde SetMatchCallback to Device PWM. Put in more page breaks between the sections to help break up the sections.
B. Fleming	7/13/2012	Updated for uEZ 2.01
B. Fleming	11/28/2012	Updated for uEZ 2.03; Remove Subsystem Device Drivers

Table of Contents

1.0 µEZ 6
1.1 Basic Types 8
2.0 Subsystem driver API 9
3.0 HAL Layer 10
3.1 HAL Registration 12
3.1.1. HALInterfaceRegister 12
3.1.2. HALInterfaceGetWorkspace 12
3.1.3. HALInterfaceFind 13
4.0 Libraries 14
4.1 Light Weight IP (lwIP) TCP/IP Stack 14
4.1.1. BSD API 15
4.1.1.1. accept 15
4.1.1.2. bind 15
4.1.1.3. close / closesocket 15
4.1.1.4. getpeername 15
4.1.1.5. getsockname 16
4.1.1.6. getsockopt 16
4.1.1.7. ioctl 16
4.1.1.8. listen 17
4.1.1.9. recv 17
4.1.1.10. recvfrom 17
4.1.1.11. select 18
4.1.1.12. send 18
4.1.1.13. sendto 18
4.1.1.14. setsockopt 19
4.1.1.15. shutdown 19
4.1.1.16. socket 19
4.1.2. Netconn API 20
4.1.2.1. netconn_accept() 20
4.1.2.2. netconn_bind() 20
4.1.2.3. netconn_close 20
4.1.2.4. netconn_connect 20
4.1.2.5. netconn_delete 21
4.1.2.6. netconn_disconnect 21
4.1.2.7. netconn_getaddr 21
4.1.2.8. netconn_gethostbyname 21
4.1.2.9. netconn_listen_with_backlog 22
4.1.2.10. netconn_recv 22
4.1.2.11. netconn_send 22
4.1.2.12. netconn_sendto 22
4.1.2.13. netconn_type 23
4.1.2.14. netconn_write 23
4.2 Simple Windowing Interface Manger (SWIM) 24
4.2.1. Concepts 24
4.2.2. Window Routines 24
4.2.2.1. swim_window_open 24
4.2.2.2. swim_window_open_noclear 25
4.2.2.3. swim_window_close 25
4.2.2.4. swim_get_horizontal_size 25
4.2.2.5. swim_get_vertical_size 25
4.2.3. Drawing Routines 26
4.2.3.1. swim_set_pen_color 26
4.2.3.2. swim_set_fill_color 26
4.2.3.3. swim_set_bkg_color 26
4.2.3.4. swim_put_pixel 26
4.2.3.5. swim_put_line 27
4.2.3.6. swim_put_box 27
4.2.3.7. swim_clear_screen 27

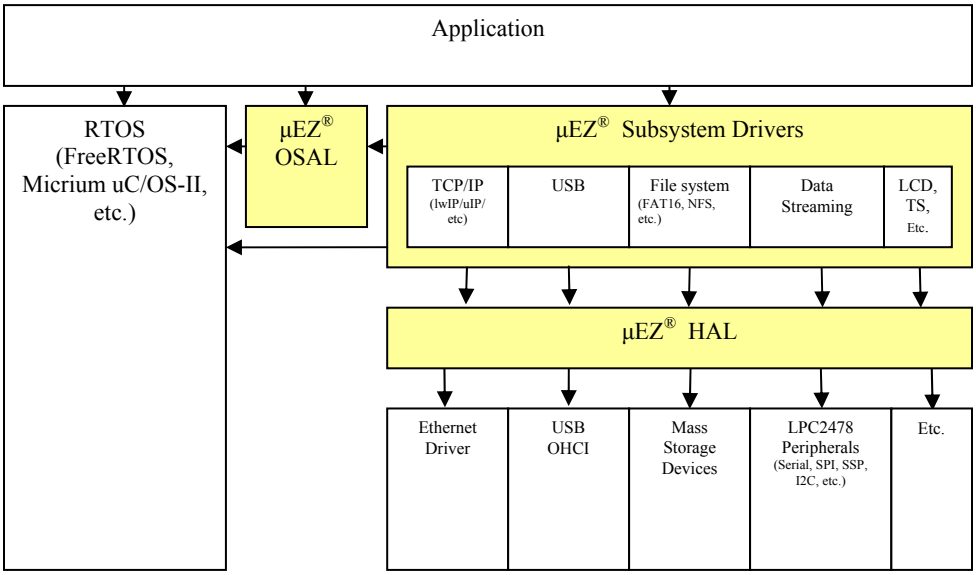
4.2.3.8. swim_put_diamond ... 28
4.2.4. Font Routines ... 28
4.2.4.1. swim_set_font ... 28
4.2.4.2. swim_get_font_height ... 28
4.2.4.3. swim_set_font_trasparency ... 28
4.2.4.4. swim_get_xy ... 29
4.2.4.5. swim_set_xy ... 29
4.2.4.6. swim_put_char ... 29
4.2.4.7. swim_put_newline ... 29
4.2.4.8. swim_put_text ... 30
4.2.4.9. swim_put_ltext ... 30
4.2.4.10. swim_put_text_xy ... 30
4.3 USB Device Generic Bulk Library ... 31
4.3.1. Concepts and Overview ... 31
4.3.1.1. On the PC Side ... 31
4.3.2. Usage ... 31
4.3.3. Library Routines ... 32
4.3.3.1. GenericBulkInitialize ... 32
4.3.3.2. GenericBulkPut ... 32
4.3.3.3. GenericBulkGet ... 32
4.3.4. More Information about libusb library ... 32
5.0 The uEZ Boot Process ... 33
6.0 Setting up a New Processor ... 34
6.1 Overview ... 34
6.2 Processor Specific RTOS ... 34
6.3 Adding uEZ Processor Device Driver Files ... 34
6.4 Processor Initialization ... 35
6.4.1. Step 1: Register the HAL Driver Interface ... 35
6.4.2. Step 2: Configure the HAL Driver ... 35
6.4.3. Miscellaneous Initialization ... 36
6.5 Interrupt Table ... 36
6.6 Testing a New Processor ... 36
7.0 Setting up a New Platform ... 37
7.1 Overview ... 37
7.2 Platform Files ... 37
7.3 Platform Configuration ... 37
7.4 Platform Initialization ... 38
7.4.1. Device Driver Registration and Initialization ... 38
7.4.1.1. Step 1: Register the Device Driver Interface ... 38
7.4.1.2. Step 2: Configure the Device Driver ... 38
7.4.2. Additional Startup Steps ... 39
7.4.3. Platform Startup Task ... 39
8.0 Appendix: Example Widget Driver Implementation ... 40
8.1 HAL Driver ... 40
8.2 Device Driver ... 42
8.3 Library ... 45
8.4 Summary ... 46

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 embedded MCU 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
- NXP LPC1768
- NXP LPC1788
- Renesas RX62N
- More processors to come …

µEZ currently supports the following RTOS families:

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

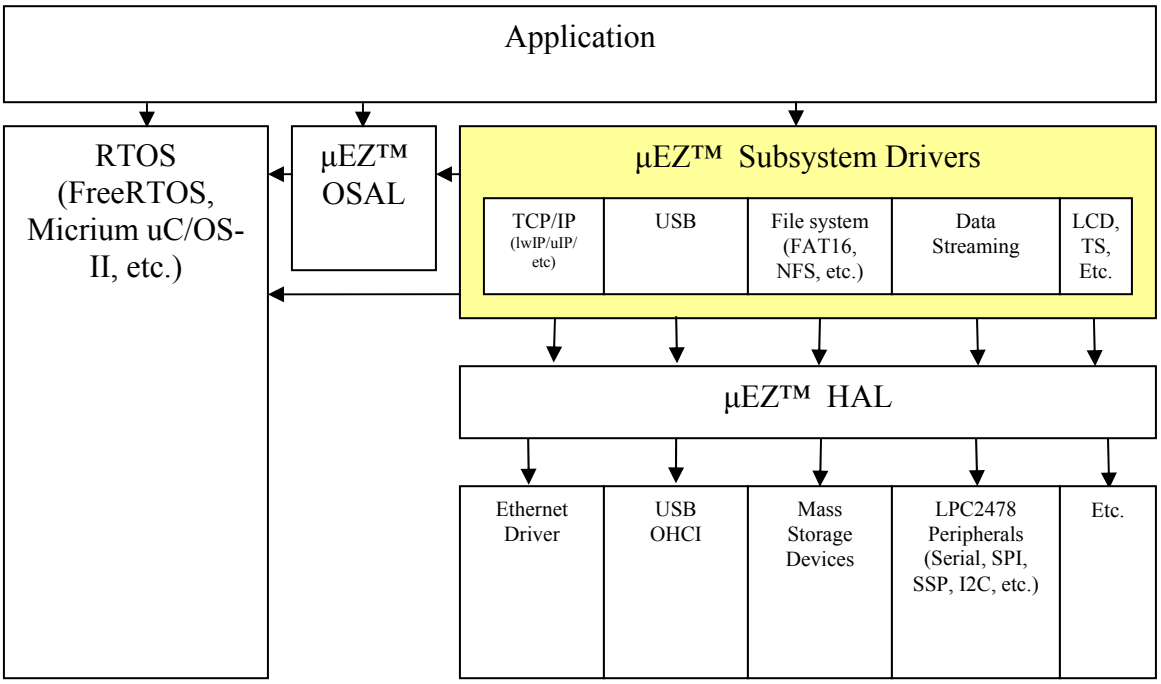
1.1 Basic Types

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.

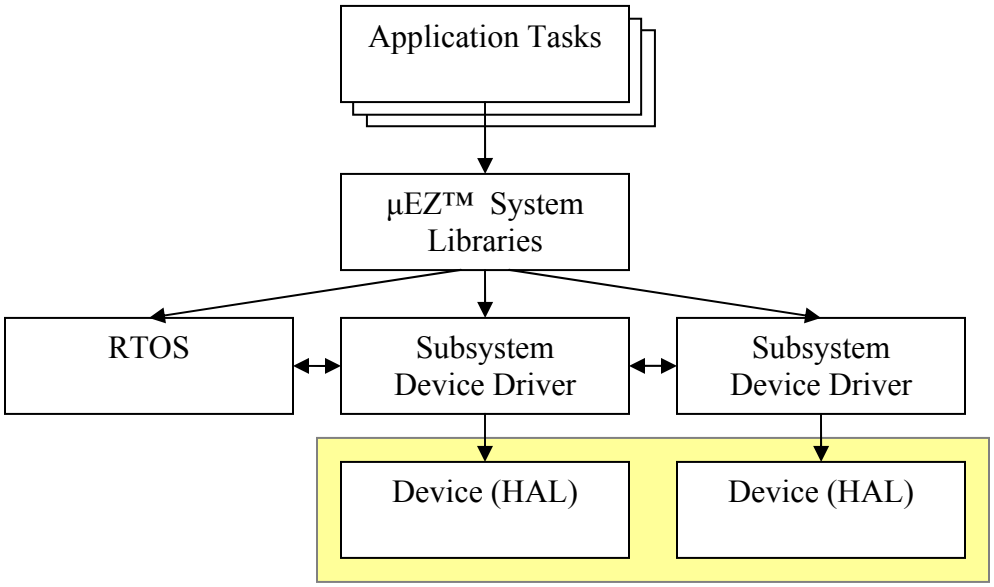
2.0 Subsystem driver API

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. See http://www.teamfdi.com/uez/docs/ (module section) for more information on the Subsystem drvier API.



3.0 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).

3.1 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.

3.1.1. 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.

3.1.2. 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.

3.1.3. 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.

4.0 Libraries

4.1 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.

4.1.1. BSD API

4.1.1.1. 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.

4.1.1.2. 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.

4.1.1.3. close / closesocket

Description:

Close a socket.

Parameters:

int s – socket to close

Results:

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

4.1.1.4. 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.

4.1.1.5. 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.

4.1.1.6. 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.

4.1.1.7. 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.

4.1.1.8. 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.

4.1.1.9. 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.

4.1.1.10. 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.

4.1.1.11. 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.

4.1.1.12. 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.

4.1.1.13. 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.

4.1.1.14. 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.

4.1.1.15. 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.

4.1.1.16. 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.

4.1.2. Netconn API

4.1.2.1. 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

4.1.2.2. 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

4.1.2.3. 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.

4.1.2.4. 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

4.1.2.5. 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

4.1.2.6. 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

4.1.2.7. 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.

4.1.2.8. 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.

4.1.2.9. 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

4.1.2.10. 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.

4.1.2.11. 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.

4.1.2.12. 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.

4.1.2.13. 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

4.1.2.14. 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.

4.2 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.

4.2.1. Concepts

4.2.2. Window Routines

4.2.2.1. 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

4.2.2.2. 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

4.2.2.3. swim_window_close

Description:

Releases a window for future use.

Parameters:

SWIM_WINDOW_T *win – Window identifier

Results:

Nothing

4.2.2.4. 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.

4.2.2.5. 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.

4.2.3. Drawing Routines

4.2.3.1. 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

4.2.3.2. 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

4.2.3.3. 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

4.2.3.4. 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

4.2.3.5. 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

4.2.3.6. 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

4.2.3.7. 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

4.2.3.8. 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

4.2.4. Font Routines

4.2.4.1. 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

4.2.4.2. 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

4.2.4.3. 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

4.2.4.4. 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

4.2.4.5. 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

4.2.4.6. 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

4.2.4.7. 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

4.2.4.8. 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

4.2.4.9. 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

4.2.4.10. 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
INT_32 x – Virtual X position of start of text
INT_32 y – Virtual Y position of start of text

Results:

Nothing

4.3 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.

4.3.1. 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.

4.3.1.1. 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.

4.3.2. 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.

4.3.3. Library Routines

4.3.3.1. 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.

4.3.3.2. 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.

4.3.3.3. 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.

4.3.4. 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.

5.0 The uEZ Boot Process

The uEZ system goes through following steps before main() is called in the application code. The following steps assume 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.

6.0 Setting up a New Processor

6.1 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.

6.2 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.

6.3 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**.

6.4 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.

6.4.1. 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.

6.4.2. 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.

6.4.3. 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.

6.5 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
```

6.6 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.

7.0 Setting up a New Platform

7.1 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.

7.2 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.

7.3 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)
```

7.4 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.

7.4.1. Device Driver Registration and Initialization

Each device peripheral is initialized using the following steps.

7.4.1.1. 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.

7.4.1.2. 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.

7.4.2. 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.

7.4.3. 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);
```

8.0 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.

8.1 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.

8.2 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.

8.3 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);
}
```

8.4 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.