
EMBER[®] APPLICATION DEVELOPMENT FUNDAMENTALS: EMBER HAL

The first half of the document describes some of the basic aspects of the Ember[®] HAL, and is recommended for anyone using EmberZNet PRO. If you need to modify the HAL or port it to a new hardware platform, you should read the entire document.

New in This Revision

Initial release (contents previously published).

Contents

1	HAL API Organization	2
2	Naming Conventions.....	2
3	API Files and Directory Structure.....	3
4	HAL API Description	3
4.1	Common Microcontroller Functions	3
4.2	Token Access and Simulated EEPROM.....	3
4.3	Peripheral Access	4
4.4	System Timer Control.....	4
4.5	Bootloading.....	4
4.6	HAL Utilities	5
4.7	Debug Channel	5
4.7.1	Virtual UART.....	5
4.7.2	Packet Trace Support.....	5
5	Customizing the HAL.....	6
5.1	Compile-Time Configuration.....	6
5.2	Custom PCBs	7
5.3	Modifying the Default Implementation.....	7

UG103.4

The Ember hardware abstraction layer (HAL) is program code between a system's hardware and its software that provides a consistent interface for applications that can run on several different hardware platforms. To take advantage of this capability, applications should access hardware through the API provided by the HAL, rather than directly. Then, when you move to new hardware, you only need to update the HAL. In some cases, due to extreme differences in hardware, the HAL API may also change slightly to accommodate the new hardware. In these cases, the limited scope of the update makes moving the application easier with the HAL than without.

The introductory parts of this document are recommended for all software developers who are using EmberZNet PRO. Developers needing to modify the HAL or port it to new a hardware platform will want to read the entire document to understand how to make changes while meeting the requirements of the EmberZNet PRO stack.

1 HAL API Organization

The HAL API is organized into the following functional sections, which are described in section 4, HAL API Description:

- **Common microcontroller functions:** APIs for control of the MCU behavior and configuration.
- **Token access:** EEPROM, Simulated EEPROM (SimEEPROM), and Token abstraction. For a detailed discussion of the token system, see document UG103.7, *Ember Application Development Fundamentals: Tokens*.
- **Peripheral access:** APIs for controlling and accessing system peripherals.
- **System timer control:** APIs for controlling and accessing the system timers.
- **Bootloading:** The use of bootloading is covered in the document UG103.6, *Ember Application Development Fundamentals: Bootloading*.
- **HAL utilities:** General-purpose APIs that may rely on hardware capabilities (for example, CRC calculation that may take advantage of hardware acceleration).
- **Debug Channel:** API traces, debugging printf's, assert and crash information, and Virtual UART support when used with a DEBUG build of the EmberZNet PRO stack.

2 Naming Conventions

HAL function names have the following prefix conventions:

- `hal`: The API sample applications use. You can remove or change the implementations of these functions as needed.
- `halCommon`: The API used by the EmberZNet PRO stack and that can also be called from an application. Custom HAL modifications must maintain the functionality of these functions.
- `halStack`: Only the EmberZNet PRO stack uses this API. These functions should not be directly called from any application, as this may violate timing constraints or cause re-entrancy problems. Custom HAL modifications must maintain the functionality of these functions.
- `halInternal`: The API that is internal to the HAL. These functions are not called directly from the stack and should not be called directly from any application. They are called only from `halStack` or `halCommon` functions. You can modify these functions, but be careful to maintain the proper functionality of any dependent `halStack` or `halCommon` functions.

Most applications will call `halXXX` and `halCommonXXX` functions and will not need to modify the HAL. If you need a special implementation or modification of the HAL, be sure to read the rest of this document as well as the data sheet for your Ember platform first.

3 API Files and Directory Structure

The HAL directory structure and files are organized to facilitate independent modification of the compiler, the MCU, and the PCB configuration.

- `<hal>/hal.h`: This master include file comprises all other relevant HAL include files, and you should include it in any source file that uses the HAL functionality. Most programs should not include lower-level includes, and instead should include this top-level `hal.h`.
- `<hal>/ember-configuration.c`: This file defines the storage for compile-time configurable stack variables and implements default implementations of functions. You can customize many of these functions by defining a preprocessor variable at compile-time and implementing a custom version of the function in the application. (For more information, see `ember-configuration-defaults.h` in the EmberZNet PRO API Reference for your platform.
- `<hal>/micro/generic`: This directory contains files used for general MCUs on POSIX-compliant systems. The default compiler is GCC.

EM250 HAL implementation

`<hal>/micro/xap2b`: This directory contains the implementation of the HAL for the XAP2b, which is the processor core used by the EM250. Functions in this directory are specific to the XAP2b but are not specific to the EM250 (see the next entry).

`<hal>/micro/xap2b/em250`: This directory implements functions that are specific to the EM250.

`<hal>/micro/xap2b/em250/board`: This directory contains header files that define the peripheral configuration and other PCB-level settings, such as initialization functions. These are used in the HAL implementations to provide the correct configurations for different PCBs.

EM35x HAL implementation

`<hal>/micro/cortexm3`: This directory contains the implementation of the HAL for the cortexm3, which is the processor core used by the EM35x. Functions in this directory are specific to the cortexm3 but are not specific to the EM35x (see the next entry).

`<hal>/micro/cortexm3/em35x`: This directory implements functions that are specific to the EM357.

`<hal>/micro/cortexm3/em35x/board`: This directory contains header files that define the peripheral configuration and other PCB-level settings, such as initialization functions. These are used in the HAL implementations to provide the correct configurations for different PCBs.

4 HAL API Description

This section gives an overview of each of the main subsections of the HAL functionality.

4.1 Common Microcontroller Functions

Common microcontroller functions include `halInit()`, `halSleep()`, and `halReboot()`. Most applications will only need to call `halInit()`, `halSleep()` (usually only ZEDs), and `halResetWatchdog()`. The functions `halInit()`, `halSleep()`, `halPowerUp()`, `halPowerDown()`, and so on call the proper functions defined in the board header file to initialize or power down any board-level peripherals.

4.2 Token Access and Simulated EEPROM

EmberZNet PRO uses persistent storage to maintain manufacturing and network configuration information when power is lost or the device is rebooted. This data is stored in tokens. A token consists of two parts: a key used to map to the physical location, and data associated with that key. Using this key-based system hides the data's

UG103.4

location from the application, which allows support for different storage mechanisms and the use of flash wear-leveling algorithms to reduce flash usage.

Note: For more information about the EmberZNet PRO token system, refer to both the token.h file and document UG103.7, *Ember Application Development Fundamentals: Tokens*.

Because the EM250, EM351, and EM357 do not contain internal EEPROM, a simulated EEPROM (also referred to as sim-eeeprom and SimEE) has been implemented to use 8 kB of upper flash memory for stack and application token storage. Because the flash cells are only qualified for up to 1,000 write cycles in the EM2xx platforms (20,000 on the EM35x platforms), the simulated EEPROM implements a wear-leveling algorithm that effectively extends the number of write cycles for individual tokens into the tens to hundreds of thousands.

The simulated EEPROM is designed to operate below the token module as transparently as possible. However, for some applications you may want to customize the behavior when a flash erase is required, because this process requires a 21 millisecond period during which interrupts cannot be serviced. You can use the

`halSimEepromCallback()` function for this purpose—while the erase must be performed to maintain proper functioning, the application can schedule it to avoid interfering with any other critical timing events. This function has a default handler implemented in the `ember-configuration.c` file that will erase the flash immediately.

Applications can override this behavior by defining

`EMBER_APPLICATION_HAS_CUSTOM_SIM_EEPROM_CALLBACK`.

A status function is also available to provide basic statistics about the usage of the simulated EEPROM. For an in-depth discussion of the simulated EEPROM, its design, its usage, and other considerations, refer to document AN703, *Using the Simulated EEPROM (EM250 and EM35x)*.

4.3 Peripheral Access

The EmberZNet PRO networking stack requires access to certain on-chip peripherals; additionally, applications may use other on-chip or on-board peripherals. The default HAL provides implementations for all required peripherals and also for some commonly used peripherals. Silicon Labs recommends that developers implement additional peripheral control within the HAL framework to facilitate easy porting and upgrade of the stack in the future.

Note: Peripheral control provided by the specific version of the EmberZNet PRO stack can be found by referring to the HAL API Reference section “Sample APIs for Peripheral Access.” An individual HAL API Reference is available in the EmberZNet PRO API reference for each Ember platform.

4.4 System Timer Control

The EmberZNet PRO stack uses the system timer to control low-resolution timing events on the order of seconds or milliseconds. High-resolution (microsecond-scale) timing is managed internally through interrupts. Silicon Labs encourages developers to use the system timer control or the event controls whenever possible; this helps to avoid replicating functionality and using scarce flash space unnecessarily. For example, you can use the function `halCommonGetInt16uMillisecondTick()` to check a previously stored value against the current value and implement a millisecond-resolution delay.

4.5 Bootloading

Bootloading functionality is also abstracted in the HAL interface. Refer to the EmberZNet PRO API reference for your platform as well document UG103.6, *Ember Application Development Fundamentals: Bootloading* for a detailed description on the use and implementation of the bootloaders.

4.6 HAL Utilities

The HAL utilities include general-purpose APIs that may rely on hardware capabilities (for example, CRC calculation that may take advantage of hardware acceleration). Crash and watchdog diagnostics, random number generation, and CRC calculation are provided by default in the HAL utilities.

4.7 Debug Channel

The EmberZNet PRO HAL implements a debug channel for communication with Ember Desktop. The debug channel provides a two-way out-of-band mechanism for the EmberZNet PRO stack and customer applications to send debugging statistics and information to Ember Desktop for large-scale analysis. It provides API traces, debugging printf's, assert and crash information, and Virtual UART support when used with a DEBUG build of the EmberZNet PRO stack. The DEBUG stack is larger than the DEBUG_OFF stack due to the debug and trace code.

Note: Three levels of builds are provided: DEBUG provides API traces for EmberZNet PRO stack APIs along with other debug capabilities. NORMAL does not provide API traces, but provides everything else. The DEBUG_OFF variant has no SerialWire interfacing whatsoever, so has no Virtual UART and no Network Analyzer event tracing other than the Packet Trace Interface (PTI), which uses the ISA3 but doesn't rely on SerialWire.

On the EM250, the SIF interface on the Packet Trace Port is used for the debug channel in addition to IDE level debugging. On the EM35x, the Serial Wire interface is used for debug channel in addition to IDE level debugging.

4.7.1 Virtual UART

EmberZNet PRO supports Virtual UART functionality with DEBUG and NORMAL builds. The Virtual UART allows normal serial APIs to still be used on the port being used by the debug channel for debug output. For the EM250, EM351, and EM357, which each only have a single physical UART numbered as port 1, the Virtual UART always occupies port 0. Virtual UART is automatically enabled when `EMBER_SERIAL0_MODE` is set to either `EMBER_SERIAL_FIFO` or `EMBER_SERIAL_BUFFER`.

When Virtual UART support is enabled, serial output sent to port 0 is encapsulated in the debug channel protocol and sent via the Packet Trace Port. The raw serial output will be displayed by Ember Desktop, and will also appear on port 4900 of the adapter. Similarly, data sent to port 4900 of the adapter will be encapsulated in the debug channel protocol and sent to the node. The raw input data can then also be read using the normal serial APIs.

The Virtual UART provides an additional port for output with debug builds that would otherwise not be available.

The following behaviors for the Virtual UART differ from normal serial UART behavior:

- `emberSerialWaitSend()` does not wait for data to finish transmitting
- `emberSerialGuaranteedPrintf()` is not guaranteed
- `EMBER_SERIALn_BLOCKING` might not block

More serial output might be dropped than normal depending on how busy the processor is with other stack functions.

4.7.2 Packet Trace Support

EmberZNet PRO supports a PacketTrace interface for use with Ember Desktop. This capability allows Ember Desktop to see all packets that are received and transmitted by all nodes in a network with no intrusion on the operation of those nodes. The PacketTrace interface works with both the dev0455 and dev0680 Ember development kit carrier boards running any application.

Custom nodes must have a Packet Trace Port to use Packet Trace functionality. In addition to the proper hardware connections to use Packet Trace functionality, the `BOARD_HEADER` must define the `PACKET_TRACE` macro. You can use the settings in `dev0455.h` for EM250 or `dev0680.h` for EM351/EM357 as a template

UG103.4

The PacketTrace interface works with both debug and non-debug builds as this support is provided by the hardware.

5 Customizing the HAL

This section describes how an end user would adapt the Silicon Labs-supplied standard HAL to their specific hardware and application requirements.

5.1 Compile-Time Configuration

The following preprocessor definitions are used to configure the EmberZNet PRO HAL. They are usually defined in the Project file, but depending on the compiler configuration they may be defined in any global preprocessor location.

Required definitions

The following preprocessor definitions must be defined:

- `PLATFORM_HEADER`: The location of the platform header file. For example, the EM357 uses `hal/micro/cortexm3/compiler/iar.h`.
- `BOARD_HEADER`: The location of the board header file. For example, the EM357 developer board uses `hal/micro/cortexm3/em35x/board/dev0580.h`. Custom boards should change this value to the new file name.
- `PLATFORMNAME`, such as `XAP2B` or `CORTEXM3`.
- `PLATFORMNAME_MICRONAME` (for example, `XAP2B_EM250` or `CORTEXM3_EM357`).
- `PHY_PHYNAME` (for example, `PHY_EM250` or `PHY_EM3XX`).
- `BOARD_BOARDNAME` (for example, `BOARD_DEV0455` or `BOARD_DEV0680`).
- `CONFIGURATION_HEADER`: Provides additional custom configuration options for `ember-configuration.c`.

Optional definitions

The following preprocessor definitions are optional:

- `APPLICATION_TOKEN_HEADER`: When using custom token definitions, this preprocessor constant is the location of the custom token definitions file.
- `DISABLE_WATCHDOG`: This preprocessor definition can completely disable the watchdog without editing code. Use this definition very sparingly and only in utility or test applications, because the watchdog is critical for robust applications.
- `EMBER_SERIALn_MODE = EMBER_SERIAL_FIFO` or `EMBER_SERIAL_BUFFER` (n is the appropriate UART port). Leave this undefined if this UART is not used by the serial driver. Note that the Buffer serial mode also enables DMA buffering functionality for the UART.
- `EMBER_SERIALn_TX_QUEUE_SIZE =` the size of the transmit queue in bytes (n is the appropriate UART port). This parameter must be defined if `EMBER_SERIALn_MODE` is defined for this UART port. In FIFO mode, the value of this definition specifies the queue size in bytes. In Buffer mode, the definition represents a queue size as a number of packet buffers, each of which is `PACKET_BUFFER_SIZE` bytes (32 bytes as of this writing).
- `EMBER_SERIALn_RX_QUEUE_SIZE =` power of 2 \leq 128 (n is the appropriate UART port). Must be defined if `EMBER_SERIALn_MODE` is defined for this UART port. This value is always quantified in bytes (even in Buffer mode).
- `EMBER_SERIALn_BLOCKING` (n is the appropriate UART port). This must be defined if this serial port uses blocking IO (note that Silicon Labs does not recommend this for most applications).

5.2 Custom PCBs

Creating a custom board is most easily done by modifying a copy of an existing board header file to match the configuration of the custom board. The board header file includes definitions for all the pinouts of external peripherals used by the HAL as well as macros to initialize and power up and down these peripherals. The board header is identified through the BOARD_HEADER preprocessor definition specified at compile time.

You can use the EM357 development kit carrier board header file dev0680.h as a template when creating a new board header. Modify the port names and pin numbers used for peripheral connections as appropriate for the custom board hardware. These definitions can usually be easily determined by referring to the board's schematic. Once the new file is complete, change the preprocessor definition BOARD_HEADER for this project to refer to the new filename.

In addition to the pinout modification, functional macros are defined within the board header file and are used to initialize, power up, and power down any board-specific peripherals. The macros are:

- `halInternalInitBoard`
- `halInternalPowerDownBoard`
- `halInternalPowerUpBoard`

Within each macro, you can call the appropriate helper `halInternal` APIs or, if the functionality is simple enough, insert the code directly.

Certain modifications might require you to change additional source files in addition to the board header. Situations that might require this include:

- Using different external interrupts or interrupt vectors
- Functionality that spans multiple physical IO ports
- Changing the core peripheral used for the functionality (for example, using a different timer or SPI peripheral)

In these cases, refer to the next section.

5.3 Modifying the Default Implementation

The functionality of the EmberZNet PRO HAL is grouped into source modules with similar functionality. These modules—the source files—can be easily replaced individually, allowing for custom implementations of their functionality. Table 1 summarizes the HAL source modules.

Table 1. EmberZNet PRO HAL Source Modules

Source Module	Description
adc	Sample functionality for accessing analog-to-digital converters built into the AVR and EM250 (refer to document AN705, <i>Using the EM250 ADC</i> , for additional information)
bootloader-interface-app	APIs for using the application bootloader
bootloader-interface-standalone	APIs for using the standalone bootloader
button	Sample functionality that can be used to access the buttons built into the development kit carrier boards
buzzer	Sample functionality that can play notes and short tunes on the buzzer built into the development kit carrier boards
crc	APIs that can be used to calculate a standard 16-bit CRC or a 16-bit CCITT CRC as used by 802.15.4
diagnostic	Sample functionality that can be used to help diagnose unknown watchdog resets and other unexpected behavior

UG103.4

Source Module	Description
flash	Internal HAL utilities used to read, erase, and write Flash in the EM250
led	Sample functionality that can be used to manipulate LEDs
mem-util	Common memory manipulation APIs such as memcpy
micro	Core HAL functionality to initialize, put to sleep, shutdown, and reboot the microcontroller and any associated peripherals
random	APIs that implement a simple pseudo-random number generator that is seeded with a true-random number when the EmberZNet PRO Stack is initialized
rc-calibrate	Sample functionality that can be used to calibrate the built-in RC oscillators of the AVR
sim-EEPROM	Simulated EEPROM system for storage of tokens in the EM250
sPI	APIs that are used to access the SPI peripherals
symbol-timer	APIs that implement the highly accurate symbol timer required by the EmberZNet PRO Stack
system-timer	APIs that implement the basic millisecond time base used by the EmberZNet PRO Stack
token	APIs to access and manipulate persistent data used by the EmberZNet PRO Stack and many applications
uart	Low-level sample APIs used by the serial utility APIs to provide serial input and output

Before modifying these peripherals, be sure you are familiar with the naming conventions and the hardware data sheet, and take care to adhere to the original contract of the function being replaced. Silicon Labs recommends that you contact Customer Support before beginning any customization of these functions to determine the simplest way to make the required changes.

CONTACT INFORMATION

Silicon Laboratories Inc.

400 West Cesar Chavez
Austin, TX 78701
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032

Please visit the Silicon Labs Technical Support web page for ZigBee products:
www.silabs.com/zigbee-support and register to submit a technical support request

Patent Notice

Silicon Labs invests in research and development to help our customers differentiate in the market with innovative low-power, small size, analog-intensive mixed-signal solutions. Silicon Labs' extensive patent portfolio is a testament to our unique approach and world-class engineering team.

The information in this document is believed to be accurate in all respects at the time of publication but is subject to change without notice. Silicon Laboratories assumes no responsibility for errors and omissions, and disclaims responsibility for any consequences resulting from the use of information included herein. Additionally, Silicon Laboratories assumes no responsibility for the functioning of undescribed features or parameters. Silicon Laboratories reserves the right to make changes without further notice. Silicon Laboratories makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Silicon Laboratories assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. Silicon Laboratories products are not designed, intended, or authorized for use in applications intended to support or sustain life, or for any other application in which the failure of the Silicon Laboratories product could create a situation where personal injury or death may occur. Should Buyer purchase or use Silicon Laboratories products for any such unintended or unauthorized application, Buyer shall indemnify and hold Silicon Laboratories harmless against all claims and damages.

Silicon Laboratories, Silicon Labs, and Ember are registered trademarks of Silicon Laboratories Inc.

Other products or brandnames mentioned herein are trademarks or registered trademarks of their respective holders.

