UG103.4: Application Development Fundamentals: HAL

Silicon Labs HAL (Hardware Abstraction Layer) 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. The HAL is designed for developers using EmberZNet PRO, EmberZNet RF4CE, and Silicon Labs Thread. This document contains new information for the Mighty Gecko (EFR32MG) family and updates for the EM3x-based MCU family of products.

Silicon Labs' *Application Development Fundamentals* series covers topics that project managers, application designers, and developers should understand before beginning to work on an embedded networking solution using Silicon Labs chips, networking stacks such as EmberZNet PRO or Silicon Labs Bluetooth Smart, and associated development tools. The documents can be used a starting place for anyone needing an introduction to developing wireless networking applications, or who is new to the Silicon Labs development environment.

KEY POINTS

- Provides information on how the HAL is organized, including naming conventions, API files, and directory structure.
- Includes an overview of each of the main subsections of the HAL functionality.
- Describes how to adapt the HAL to your specific hardware and application requirements.

1. Introduction

The 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, EmberZNet RF4CE, or Silicon Labs Thread. 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 networking stack.

2. HAL API Organization

The HAL API is organized into the following functional sections, which are described in section 5. 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: 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: 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 printfs, assert and crash information, and Virtual UART support when used with a DEBUG build of the stack.

3. 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 stack and that can also be called from an application. Custom HAL modifications must maintain the functionality of these functions.
- `halStack`: Only the 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 Silicon Labs platform first.

4. 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 API Reference for your software).
- `<hal>/micro/generic`: This directory contains files used for general MCUs on POSIX-compliant systems. The default compiler is GCC.

4.1 HAL Implementation for ARM Cortex-M3 SOC Platforms

There is a HAL implementation for the ARM Cortext-M3 System on Chip (SOC) platforms as follows:

- `<hal>/micro/cortexm3`: This directory contains the implementation of the HAL for the cortexm3, which is the processor core used by the EM3x and EFR32 platforms. Functions in this directory are specific to the cortexm3 but are not specific to a particular microcontroller family or variant (see the next entry).
- `<hal>/micro/cortexm3/{mcu_family}`: This directory implements functions that are specific to the a particular MCU family, such as `<hal>/micro/cortexm3/em35x` for the EM3x-based MCU family, including variants such as the EM357, EM3588, and EM346; or `<hal>/micro/cortexm3/efm32` for the EFR32MG.
- `<hal>/micro/cortexm3/bootloader`: This directory implements functions that pertain to the on-chip bootloaders used on Cortex M3-based platforms to facilitate runtime loading/updating of applications. (More MCU-specific files can be found in `<hal>/micro/cortexm3/{mcu_family}/bootloader`.)
- `<hal>/micro/cortexm3/{mcu_family}/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.

5. HAL API Description

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

5.1 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.

5.2 Token Access and Simulated EEPROM

The networking stack 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 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 Silicon Labs token system, refer to both the `token.h` file and document *UG103.7: Application Development Fundamentals: Tokens*.

Because EM3x and EFR32 process technology does not offer an internal EEPROM, a simulated EEPROM (also referred to as sim-eeprom and SimEE) is implemented to use a section of internal flash memory for stack and application token storage. Parts that use the simulated EEPROM to store non-volatile data have different levels of flash performance with respect to guaranteed write cycles. (See the Reference Manual for your specific MCU for details about minimum flash endurance expected over lifetime and temperature variations.). Recently, version 2 of the simulated EEPROM has been released. For version 1, the ARM Cortex MCUs utilize either 4 kB or 8 kB (default) of upper flash memory to store the simulated EEPROM. For version 2, the simulated EEPROM requires 36 kB of upper flash storage. Due to the limited write cycles, the simulated EEPROM implements a wear-leveling algorithm that effectively extends the number of write cycles for individual tokens.

The simulated EEPROM is designed to operate below the token module as transparently as possible. The application is only required to implement one callback and periodically call one utility function. In addition, a status function is available to provide the application with two basic statistics about simulated EEPROM usage.

Erasure of flash pages is under the application's control because erasing a page will prevent interrupts from being serviced for 21 ms. 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 for the EM35x and Mighty Gecko (EFR32MG) SoC Platforms*.

5.3 Peripheral Access

The 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 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 API reference for each Silicon Labs platform.

5.4 System Timer Control

The networking 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.

5.5 Bootloading

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

5.6 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.

5.7 Debug Channel

The networking stack HAL implements a debug channel for communication with Simplicity Studio. The debug channel provides a two-way out-of-band mechanism for the stack and customer applications to send debugging statistics and information to Simplicity Studio for large-scale analysis. It provides API traces and debugging printfs in a Debug build of the stack, as well as assert and crash information and Virtual UART support when used with a Normal build of the stack.

There are three build levels of the stack:

- Full Debug provides API traces for stack APIs along with other debug capabilities.
- Normal does not provide API traces or `emberDebugPrintf()` support, but provides everything else.
- Debug Off has no SerialWire interfacing whatsoever, so has no Virtual UART and no Network Analyzer event tracing other than the Packet Trace Interface (PTI). PTI uses a debug adapter such as the ISA3 or Wireless Starter Kit (WSTK) but doesn't rely on SerialWire. On the ARM Cortex platforms, the Serial Wire interface is used for debug channel in addition to development environment-level debugging.

  Note: The Debug level is larger than the Normal level due to additional features.

5.7.1 Virtual UART

Some networking stacks support Virtual UART (VUART) functionality as part of their core stack libraries, or through a separate library that provides basic debugging capabilities, such as the Debug Basic Library found in EmberZNet's ZCL Application Framework. VUART, also called "two-way debug", allows normal serial APIs to still be used on the port being used by the debug channel (SerialWire interface) for debug input and output. When enabled,Virtual UART is designated in software as serial port 0. VUART functionality is available for SoC platforms only and is enabled when `EMBER_SERIAL0_MODE` is set to either `EMBER_SERIAL_FIFO` or `EMBER_SERIAL_BUFFER`.

Note: Virtual UART is different from the virtual COM port support, known as "VCOM", provided in the EFR32 Board Support Package (BSP) via `com_device.h`. VCOM routes the physical serial port connections of USART0 back through the WSTK's onboard TTL-to-USB converter for use as a communications port by a USB host. VCOM and VUART are independent of one another and can be enabled separately, together, or not at all.

When VUART support is enabled, serial output sent to port 0 is encapsulated in the debug channel protocol and sent bidirectionally via the SWO and SWDIO lines through the debug adapter (ISA3 or WSTK). The raw serial output will be displayed by Simplicity Studio's Device Console tool, and will also appear on port 4900 of the debug 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.

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

The following behaviors for VUART 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.

5.7.2 Packet Trace Support

The networking stack supports a PacketTrace interface (PTI) for use with Simplicity Studio. This capability allows Simplicity Studio 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 the radio modules supplied in the development kits but can also be enabled on custom hardware designs.

Custom nodes must have a Packet Trace Port (part of the Mini-Simplicity Connector interface on EFR32-based designs) to use Packet Trace functionality. In addition to the proper hardware connections (such as PTI_FRAME/PTI_DATA for EM3x-based designs or FRC_DFRAME/FRC_DATA for EFR32-based designs) to use Packet Trace functionality, the BOARD_HEADER must define the PACKET_TRACE macro. You can use the settings in your provided board header file (generated by the Simplicity Studio's IDE with a `_board.h` suffix) as a template

PTI is available regardless of the Debug level chosen for the build because this support is provided by the hardware. Disabling this interface requires that the PTI_FRAME and PTI_DATA pins be assigned to different GPIO functions. For more information on Board Header Files, see section 6.2 Custom PCBs.

6. Customizing the HAL

This section describes how to adapt the Silicon Labs-supplied standard HAL to your specific hardware and application requirements.

6.1 Compile-Time Configuration

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

6.1.1 Required Definitions

The following preprocessor definitions must be defined:

- `PLATFORM_HEADER`: The location of the platform header file. For example, the EM3588 uses `hal/micro/cortexm3/compiler/iar.h`.
- `BOARD_HEADER`: The location of the board header file. For example, the EM3588 developer board uses `hal/micro/cortexm3/em35x/board/dev0680etm.h`. Custom boards should change this value to the new file name.
- `PLATFORMNAME` (for example, CORTEXM3).
- `PLATFORMNAME_MICRONAME` (for example, CORTEXM3_EM3588).
- `PHY_PHYNAME` (for example PHY_EFR32).
- `BOARD_BOARDNAME` (for example, BOARD_BRD4151A or BOARD_DEV0680).
- `CONFIGURATION_HEADER`: Provides additional custom configuration options for `ember-configuration.c`.

6.1.2 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.

For EM3x:

- `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 up to 16383. 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` = size of the receive queue in bytes (`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) and may be up to 16383.
- `EMBER_SERIALn_BLOCKING` (`n` is the appropriate UART port). This must be defined if this serial port uses blocking IO.

For EFR32:

- `COM_USARTn_ENABLE` (`n` is the appropriate UART port). This enables writing and reading from the specified UART. This is typically defined in `com_device.h`.
- `COM_n_RX_QUEUE_SIZE` = size of the receive queue in bytes (`n` is the appropriate UART port). If undefined, the default is 64 bytes.
- `COM_n_TX_QUEUE_SIZE` = size of the transmit queue in bytes (`n` is the appropriate UART port). If undefined, the default is 128 bytes.
- `COM_USARTn_HW_FC` or `COM_USARTn_SW_FC` (`n` is the appropriate UART port). This enables hardware or software flow control, respectively.

6.2 Custom PCBs

Creating a custom GPIO configuration for your target hardware is most easily done by using the Simplicity Studio IDE to generate a board header file based on a copy of an existing board header file and then editing the generated 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. Board header files generated by the IDE will typically have a `_board.h` suffix, but several pre-made board header files for specific reference boards are available in the `hal/micro/cortexm3/{mcu_family}/board` folder of your stack release.

Modify the port names and pin numbers (and potentially location numbers in the case of EFR32) 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 section 6.3 Modifying the Default Implementation.

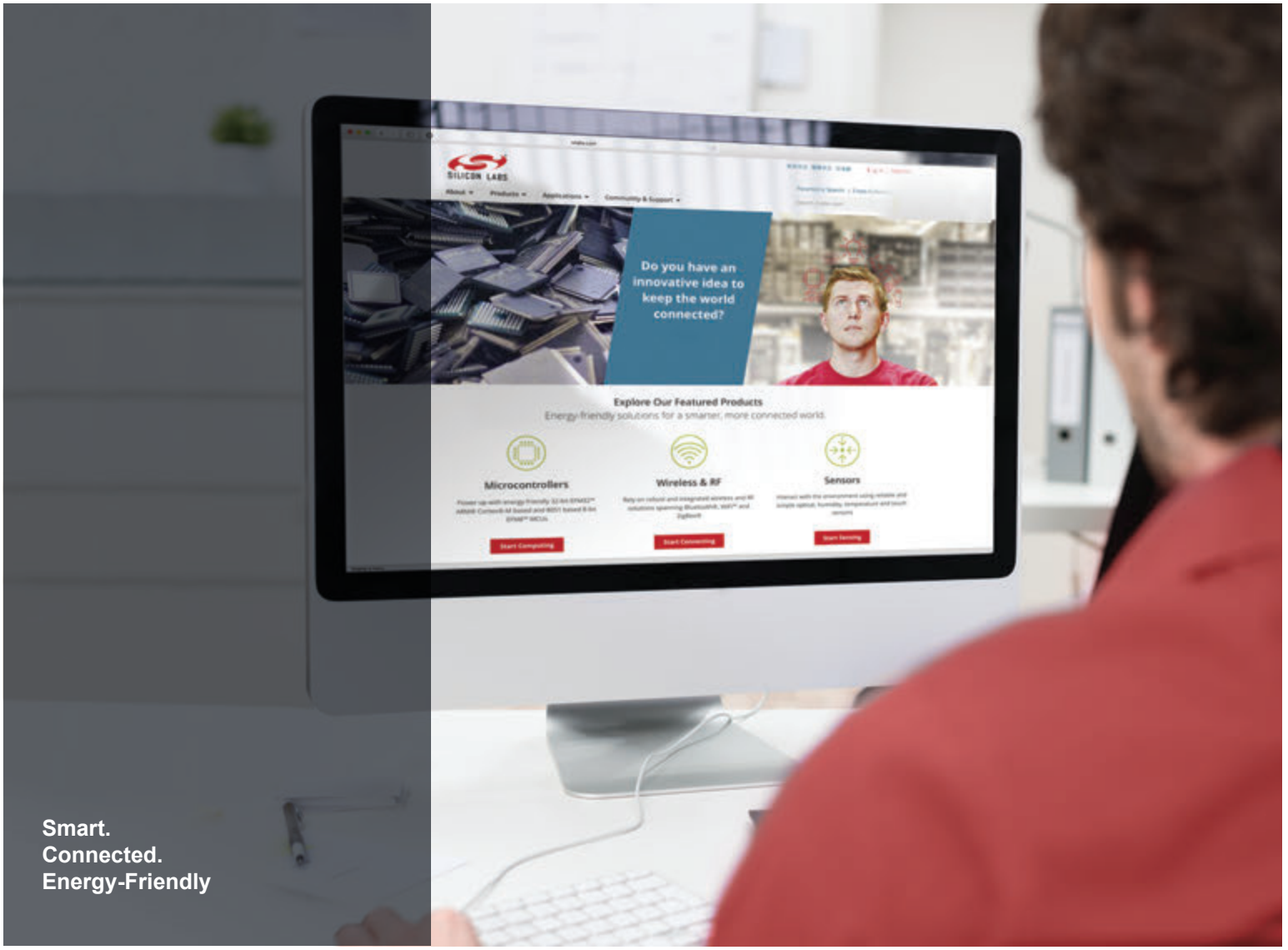
6.3 Modifying the Default Implementation

The functionality of the networking stack 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. The following table summarizes the HAL source modules.

Table 6.1. HAL Source Modules

Source Module	Description
Adc	Sample functionality for accessing analog-to-digital converters built into the SoC (refer to document *AN715: Using the EM35x ADC* 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.
Flash	Internal HAL utilities used to read, erase, and write Flash in the SoC.
Led	Sample functionality that can be used to manipulate LEDs.
mem-util	Common memory manipulation APIs such as memcopy.
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 stack is initialized.
sim-eeprom	Simulated EEPROM system for storage of tokens in the SoC.
Spi	APIs that are used to access the SPI peripherals.
symbol-timer	APIs that implement the highly accurate symbol timer required by the stack.
system-timer	APIs that implement the basic millisecond time base used by the stack.
Token	APIs to access and manipulate persistent data used by the 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.



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS®

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com