AN0511

CS40L25 MCU Driver User Guide

Introduction

This user guide details the design and use of the MCU Driver Software Package, specifically the CS40L25 MCU driver for the CS40L25 Haptic Amplifier.

The MCU Driver Software Package is a package of C driver code, tools, and example code to help users integrate the CS40L25 into their system. The CS40L25 MCU driver is intended to provide the user a set of tools and APIs to help make interacting with the CS40L25 simpler. The MCU Driver Software Package can be acquired by contacting your Cirrus Logic Representative, or directly from the Cirrus Logic GitHub repository via the following command:

```
git clone https://github.com/CirrusLogic/mcu-drivers.git
```

This guide is primarily intended for those involved in end system implementation, integration, and testing, who will use the MCU Driver Software Package to integrate the CS40L25 MCU driver source code into the end system's host MCU software. After reviewing this guide, the reader will be able to begin software integration of the CS40L25 MCU driver and then have the ability to initialize, reset, boot, configure, and service events from the CS40L25. Specific system configurations and use-cases are not addressed in this guide. For information on configuring the device for a specific use-case, please refer to the CS40L25 silicon data sheet, or contact a Cirrus Logic representative.

This guide should be used along with the following materials:

- The driver source code contained in the MCU Driver Software Package release v4.1.0
- CS40L25 silicon data sheet
- CS40L25 firmware release notes

For any questions regarding this guide, the MCU Driver Software Package, or CS40L25 system integration, please contact your Cirrus Logic representative.

Table of Contents

1 Package Description

The following sections detail the contents of the CS40L25 MCU Driver Software Package.

1.1 Filesystem Layout

Once the MCU Driver Software Package archive ZIP file is decompressed, the folder structure created will resemble the table below. A description is given of the source code contents of each folder/subfolder.

Table 1 Filesystem Layout Table

File/Folder	Source Code Contents
/common/	Modules common to all drivers
/common/system_test_hw_0/	Board Support Package (BSP) implementation for ST NUCLEO-F401RE
/cs40l25/	Primary driver source/makefile
/cs40l25/artifact_info.txt	Contains tracking information such as Git SHA, CI build job name and number, tools versions, artifact statistics
/cs40l25/baremetal/	`main.c` for Baremetal example project
/cs40l25/bsp/	Haptics-related Board Support Package (BSP) API that abstracts the CS40L25 driver calls to the example projects
/cs40l25/config/	Various C files, headers, and WISCE™ scripts that should be updated by the user to tailor the driver configuration to be more suitable for their system
/cs40l25/freertos/	`main.c` for FreeRTOS example project
/cs40l25/fw_<x>_<y>_<z>/	Source code for arrays containing CS40L25 Halo Core™ DSP firmware and coefficient payloads firmware release x.y.z
/third_party/arm/	Location to add ARM CMSIS
/third_party/st/	Location to add STM32CubeF4 - ST MCU HAL/Driver SDK for STM32F4-based MCUs
/tools/firmware_converter/	Python tool for converting CS40L25 Halo Core™ DSP `.WMFW` firmware files and `.BIN` coefficient files to fw_img_v2 format. See Firmware Converter
/tools/wisce_script_converter/	Python tool for converting WISCE scripts for CS40L25 hardware configuration to driver source files `cs40l25_syscfg_regs.h/.c`. See WISCE™ Script Converter
/doc.zip	Doxygen HTML-based source code documentation for the driver library source code listed in Driver Source File Description
/release_notes.txt	Release notes for the current and past releases of the MCU Driver Software Package

1.2 Driver Source File Description

Table 2 gives a description of each of the source files used to implement the CS40L25 MCU driver. Please note these are only the source files required for the library and does not include the following:

- C standard library headers
- Board Support Package (BSP) implementation source files
- Any application layer requirements for a full system implementation. A full system implementation can be found by investigating the dependencies of either the 'baremetal' or the 'freertos' example projects

Table 2 Driver Source File Description Table

Folder	Filename	Description
/cs40l25/	cs40l25.h cs40l25.c	Implementation of CS40L25 MCU driver, including all public API and supporting functions
	cs40l25_ext.h cs40l25_ext.c	Implementation of the CS40L25 MCU driver extended API
	cs40l25_spec.h cs40l25_spec.c	All constants and arrays exported directly from the CS40L25 silicon data sheet required by the driver
	cs40l25_sym.h cs40l25_cal_sym.h	Defines for the fw_img_v2 Symbol ID table for both normal (`cs40l25_sym.h`) and calibration firmware loads (`cs40l25_cal_sym.h`)

Folder	Filename	Description
	makefile	Makefile for all CS40L25-related build targets, including: 'driver_lib_cm4', 'baremetal', 'freertos'
	version.h	Macros for release version
/cs40l25/config/	cs40l25_syscfg_regs.h cs40l25_syscfg_regs.c	Defines and array for the driver system configuration register set. Note: These files are automatically generated via `wisce_script_converter.py` when executing the makefile.
/common/	bsp_driver_if.h	Definition of the system BSP-to-Driver Interface, defining all types and functions the Board Support Package (BSP) will implement to support a device driver.
	fw_img.c fw_img.h	fw_img parsing code, which can be used to provide the CS40L25 MCU driver firmware
/	sdk_version.h	Macros for MCU Driver Software Package release version

2 Integration Guide

The following section outlines steps needed to quickly get started integrating the CS40L25 MCU driver into a new system. Integration of the CS40L25 MCU driver into an end system host MCU software framework will entail the requirements and steps in the following sections.

Integration of the CS40L25 MCU driver into an end system host MCU software framework will require the following steps:

1. Implement the functions required of the BSP to support the driver—the BSP-to-Driver Interface.
2. Generate the base system configuration files `cs40l25/cs40l25_syscfg_regs.h/.c.`
3. Convert the Halo Core™ DSP firmware and coefficient/tuning files to a fw_img_v2 block.
4. Call the CS40L25 MCU driverPublic API.

2.1 Implement BSP-to-Driver Interface

The first step to integrating the CS40L25 MCU driver is to implement the BSP-to-Driver Interface. Implementing this interface is outlined in the sections below.

2.1.1 Implement Interface Functions

Implement the interface functions described in BSP-to-Driver Interface required for the end system. Not all functions will be required—please note which functions must be implemented for a specific configuration of the CS40L25 MCU driver.

The CS40L25 MCU driver accesses the BSP-to-Driver interface functions via the table of function pointers declared in the interface handle `bsp_driver_if_g` found in `common/bsp_driver_if.h`. This table of function pointers must be defined, and the handle `bsp_driver_if_g` must be defined and assigned the address of the instance of the interface (for e.g., see `bsp_driver_if_g` and `bsp_driver_if_s` in `common/system_test_hw_0/hw_0_bsp.c`).

2.1.2 Implement cs40l25_bsp_config_t Members

In the BSP-to-Driver interface implementation, the constants, functions, and function pointers in the table below used in configuring the driver (see Allocate Driver Resources) must be implemented.

Table 3 cs40l25_bsp_config_t Members Table

cs40l25_bsp_config_t Member	Description
bsp_dev_id	A unique identifier for the specific device being integrated. This will be passed back to the BSP through the BSP-to-Driver Interface, typically during I2C or SPI transaction requests, to allow the BSP to uniquely identify the device and ensure the correct control bus is used.
bsp_int_gpio_id	An identifier for the BSP to use when registering the callback function for the interrupt handler relating to the IRQ pin on the device. Used for calls to `register_gpio_cb`.
bsp_reset_gpio_id	An identifier for the BSP to use when toggling the reset line to the device. Used for calls to `set_gpio`.
bus_type	Whether the device is connected to a SPI (`BSP_BUS_TYPE_SPI`) or I2C (`BSP_BUS_TYPE_I2C`) control bus.
notification_cb	A function pointer to a function that the user would like to be called in the event of an IRQ.

cs40l25_bsp_config_t Member	Description
	The function should be of type: `void (*cs40l25_notification_callback_t)(uint32_t event_flags, void *arg);`
notification_cb_arg	An argument that will be passed back to the BSP when `notification_cb` is called, through the `arg` parameter.

An example of a Board Support Package (BSP) implementation used for development and testing of the driver can be found in the following files:

- `common/system_test_hw_0/hw_0.bsp.h`
- `common/system_test_hw_0/hw_0.bsp.c`

2.2 Generate Base Configuration

The CS40L25 MCU driver build is dependent on the system configuration files `cs40l25/config/cs40l25_syscfg_regs.h` and `cs40l25/config/cs40l25_syscfg_regs.c`. These files are generated by code generation tools, typically during the build using the `wisce_script_converter` tool and a WISCE™ script that has been generated during any prototyping of the end system. The registers written by the `cs40l25_syscfg_regs` should be those common to all use-cases in the system and will be written out to the device during the initial device probe in `cs40l25_boot`.

An example call to this tool can be found in `cs40l25/makefile` at the make target `wisce_script_converter`. This call can take the form below.

Call to wisce_script_converter.py

```
~/mcu-drivers/cs40l25$ python3 ../../tools/wisce_script_converter/wisce_script_converter.py -c
c_array -p cs40l25 -i wisce_init.txt

wisce_to_syscfg_reg_converter
Convert from WISCE Script Text file to Alt-OS Syscfg Reg
Version 1.0.0

Command: c_array
WISCE script path: wisce_init.txt
Output path: .
Exported to:
./cs40l25_syscfg_regs.h
./cs40l25_syscfg_regs.c

Exit.
```

The output `cs40l25_syscfg_regs.h` and `cs40l25_syscfg_regs.c` will then be used for building and linking the final driver into the system firmware.

2.3 Convert a WMFW and WMDRs to fw_img

To take advantage of a CS40L25 Halo Core™ DSP firmware, the conversion of Halo Core™ DSP firmware and coefficient/config files to a fw_img_v2 block is required. The fw_img_v2 data contains all the information required to load firmware and discover what algorithms and controls are available on it, and is also flexible enough to support multiple firmware and configuration use cases. For more information on the contents of the fw_img_v2 block, please see Appendix A—fw_img_v2 Specification. After conversion, the fw_img_v2 block can be integrated into the end system either as:

- A C source file array that is compiled with the system firmware.
- A binary image that can be programmed to any memory—host MCU ROM or ROM external to the MCU such as

serial EEPROM or flash.

Conversion of firmware and coefficient/config files is done using the Python utility in `tools/firmware_converter`. The command line call below demonstrates how the `firmware_converter` utility was used to convert a `.WMFW` file into the files `cs40l25/fw_8_12_2/clab/cs40l25_fw_img.h` and `cs40l25/fw_8_12_2/clab/cs40l25_fw_img.c` used for the CS40L25 example projects. See the `cs40l25/makefile` for more details.

```
~/mcu-drivers/cs40l25/fw_pt$ python3 ../../../tools/firmware_converter/firmware_converter.py
fw_img_v2 cs40l25 ../prince_haptics_ctrl_ram_remap_clab_0A0603.wmfw --
wmdr ../default_clab.bin ../dvl.bin --sym-input ../../cs40l25_sym.h --generic-sym

firmware_converter
Convert from WMFW/WMDR ("BIN") Files to C Header/Source
Version 3.2.0

Command: fw_img_v2
Part Number: cs40l25
WMFW Path: ../prince_haptics_ctrl_ram_remap_clab_0A0603.wmfw
WMDR Path: ../default_clab.bin
WMDR Path: ../dvl.bin
No suffix
Input Symbol ID Header: ../../cs40l25_sym.h
Exported to files:
cs40l25_fw_img.h
cs40l25_fw_img.c

Exit.
```

The arguments used for the call to the tool are broken down in the table below. For further details on these and other options when using `firmware_converter`, please see Firmware Converter.

Table 4 firmware_converter Arguments

Argument	Notes
fw_img_v2	The 'command' to the tool is to convert the `.WMFW` and `.BIN` file to fw_img_v2 block format.
cs40l25	The CS40L25 is specified to use the correct hardware memory map.
../prince_haptics_ctrl_ram_remap_clab_0A0603.wmfw	Specify the `.WMFW` file.
--wmdr ../default_clab.bin ../dvl.bin	Specify the firmware coefficient and configuration files.
--sym-input ../../cs40l25_sym.h	Specify the input Symbol ID header file path.
--generic-sym	Use "generic" names for the Symbol ID defines that correspond to the controls in the primary firmware algorithm.

2.4 Call the Driver Public APIs

The last step in integration of the MCU Driver Software Package is to call the Driver Public API in any application layer or hardware abstraction layer. Calling the Driver Public API requires first allocating and defining external resources the driver needs, and then calling the API in the proper order.

2.4.1 Allocate Driver Resources

The following resources must be defined to be used for driver initialization, configuration, and use. The 'Scope' should be noted as some maybe file scope while others are only function scope. References to the filename and symbol for the example implementations are included.

Table 5 Allocated Driver Resources Table

C Type	Description	Notes	Scope	Board Support Package (BSP) Implementation Filename	Example Symbol Name
cs40l25_t	Driver state data structure	The `cs40l25_initialize` call will initialize the data structure to a known state.	file	cs40l25/bsp/hw_0_bsp_cs40l25.c	cs40l25_driver
cs40l25_notification_callback_t	Driver-to-BSP Notification Callback	Called when any important driver events occur	file	common/system_hw_0_bsp/hw_0_bsp.c	bsp_notification_callback
cs40l25_config_t	Driver configuration data structure	This contains all configuration pertaining to system hardware configuration and BSP configuration. It is important to note that the calibration parameters obtained from the calibration sequence are included in this structure as member `cal_data`, which is type `cs40l25_calibration_t`.	function	cs40l25/bsp/hw_0_bsp_cs40l25.c	haptic_config
fw_img_boot_state_t	Data structure to describe Halo Core™ DSP firmware and coefficient download	Contains the information for the most recently loaded fw_img block.	file	cs40l25/bsp/hw_0_bsp_cs40l25.c	boot_state

2.4.2 Driver Public API Call Ordering

While the calls to the Driver Public APIs can be organized in various ways, the simplified ordering below gives an example that will result in successful calibration and final power up with the device ready to process haptic events. To further illustrate the example, cross-references to the calling functions from `cs40l25/bsp/hw_0_bsp_cs40l25.c` are given.

Power Cycle 1—Calibration.

Table 6 Calibration Power Cycle API Calls

Caller from Board Support Package (BSP)	Driver Public API Call	Notes
bsp_dut_initialize()	cs40l25_initialize()	
bsp_dut_initialize()	cs40l25_configure()	
bsp_dut_reset()	cs40l25_reset()	
bsp_dut_power_down()	cs40l25_power(CS40L25_POWER_DOWN)	Exit Basic Haptics Mode (BHM)
bsp_dut_boot(true)	cs40l25_boot() fw_img_read_header() fw_img_process() cs40l25_write_block()	
bsp_dut_power_up()	cs40l25_power(CS40L25_POWER_UP)	

Caller from Board Support Package (BSP)	Driver Public API Call	Notes
bsp_dut_calibrate()	cs40l25_calibrate()	If the Calibration sequence was successful, then: • `cs40l25_driver.config.cal_data` members `is_valid_f0` and `is_valid_qest` should be true • `cs40l25_driver.config.cal_data` will contain the calibration parameters • The system firmware should save these parameters to non-volatile memory to be applied to the configuration data at the next power cycle

Power Cycle 2—e.g., Halo Core™ DSP Firmware with CLAB algorithm.

Table 7 Normal Power Cycle API Calls

Caller from Board Support Package (BSP)	Driver Public API Call	Notes
bsp_dut_initialize()	cs40l25_initialize()	
bsp_dut_initialize()	cs40l25_configure()	If a previous Calibration sequence was run and valid calibration parameters were retrieved from non-volatile memory, then the retrieved parameters should be applied to `cs40l25_config_t.cal_data`, ensuring that both `cal_data` members `is_valid_f0` and `is_valid_qest` are set to `true`.
bsp_dut_reset()	cs40l25_reset()	
bsp_dut_power_down()	cs40l25_power(CS40L25_POWER_DOWN)	Exit Basic Haptics Mode (BHM)
bsp_dut_boot(false)	cs40l25_boot() fw_img_read_header() fw_img_process() cs40l25_write_block()	
bsp_dut_power_up()	cs40l25_power(CS40L25_POWER_UP)	After this call, haptics process can be performed as the Halo Core™ DSP Firmware has been configured.
bsp_dut_hibernate()	cs40l25_power(CS40L25_POWER_HIBERNATE)	

2.4.3 Direct Calls to Register Manipulation Functions

The CS40L25 driver provides various functions for accessing registers on the device. Examples of using these are shown below, and in the BSP example code `cs40l25/bsp/hw_0_bsp_cs40l25.c`.

Reading and Writing Hardware and Firmware Registers

```
// Read register 0x00000000 (DEVID)
ret = cs40l25_read_reg(&cs40l25_driver, 0x0, &val);
if (ret)
  return -1;

// Write register 0x00002014 (GLOBAL_ENABLES)
ret = cs40l25_write_reg(&cs40l25_driver, 0x2014, 0x1);
if (ret)
  return -1;

// In register 0x00006000 (AMP_CTRL), set the AMP_VOL_PCM bitfield (bits 13:3) to mute (0x400)
ret = cs40l25_update_reg(&cs40l25_driver, 0x6000, 0x00003FF8, (0x400 << 3));
if (ret)
  return -1;

// Look up the register address of a symbol on a running or preloaded firmware
uint32_t sym_addr = cs40l25_find_symbol(&cs40l25_driver, 1,
CS40L25_SYM_VIBEGEN_COMPENSATION_ENABLE);
if (!sym_addr)
  return -1;

// Read the current value of that address
ret = cs40l25_read_reg(&cs40l25_driver, sym_addr, &val);
if (ret)
  return -1;

// Write a value to that address
ret = cs40l25_write_reg(&cs40l25_driver, sym_addr, 0x3);
if (ret)
  return -1;
```

2.4.4 Processing a fw_img Block

The fw_img block generated from the `firmware_converter` tool can be processed using the fw_img module provided in `common/fw_img.c` and `common/fw_img.h`.

There are two Driver Public API functions that need to be used to decode the fw_img file:

```
extern uint32_t fw_img_read_header(fw_img_boot_state_t *state);
extern uint32_t fw_img_process(fw_img_boot_state_t *state);
```

and two Driver Public API functions that are used to feed the resulting output to the driver:

```
uint32_t cs40l25_write_block(cs40l25_t *driver, uint32_t addr, uint8_t *data, uint32_t length);
uint32_t cs40l25_boot(cs40l25_t *driver, uint32_t dsp_core, fw_img_info_t *fw_info);
```

An example of the process for decoding fw_img blocks can be found in `cs40l25/bsp/hw_0_bsp_cs40l25.c : bsp_dut_boot()`, however the basic process should look as follows:

1. Allocate a new `fw_img_boot_state_t` struct and initialize to zero (henceforth referred to as `boot_state`)
2. If you are pulling the fw_img from external flash, pull the fw_img_v2 header (40 bytes) into memory.
3. Initialize:
   a. `boot_state.fw_img_blocks` to point at the start of the fw_img data.
   b. `boot_state.fw_img_blocks_size` to the size of the currently available data.
4. Call `fw_img_read_header(&boot_state)` If this function returns `FW_IMG_STATUS_OK`, then the fw_img_v2 header will be copied into a `fw_img_v2_header_t` struct (`boot_state.fw_info.header`) and this can be used to derive the total size of the fw_img_v2 data.
5. Three data blocks must now be allocated and assigned to `boot_state`:
   a. `boot_state.fw_info.sym_table` should be set to a memory allocation of size: `boot_state.fw_info.header.sym_table_size * sizeof(fw_img_v1_sym_table_t)`
   b. `boot_state.fw_info.alg_id_list` should be set to a memory allocation of size: `boot_state.fw_info.header.alg_id_list_size * sizeof(uint32_t)`
   c. `boot_state.block_data` should be set to a memory allocation of the maximum block size: `boot_state.fw_info.header.max_block_size`
   d. `boot_state.block_data_size` should be set to: `boot_state.fw_info.header.max_block_size`
6. The rest of the fw_img_v2 data can then be processed by making repeated calls to `fw_img_process(&boot_state)` and handling the return code as follows:
   a. `FW_IMG_STATUS_DATA_READY`—data is ready to be passed to the CS40L25 driver. Accordingly, the `cs40l25_write_block` function should be called: `cs40l25_write_block(&cs40l25_driver, boot_state.block.block_addr, boot_state.block_data, boot_state.block.block_size);`
   b. `FW_IMG_STATUS_NODATA`—the current fw_img_v2 data has been processed, and the fw_img module is ready for more to be pulled from the external flash. Update `boot_state.fw_img_blocks` and `boot_state.fw_img_blocks_size` to point at new fw_img data.
   c. `FW_IMG_STATUS_OK`—the fw_img data has been completely processed and you can move on to boot the CS40L25.
7. The fw_img_v2 data has now been processed, which has done four things:
   a. Extracted the firmware's symbol table (used for looking up coefficient memory addresses within the firmware).
   b. Extracted the algorithm ID list (used for looking up which algorithms are present in the firmware).
   c. Extracted the fw_img_v2 header (which contains the fw id and revision).
   d. Written out the firmware's contents to Halo Core™ DSP memory.
8. The memory allocated for `boot_state.block_data` is now no longer required and can be freed.
9. Finally, the data about the firmware can be passed to the driver and to get the CS40L25 ready to be powered up: `cs40l25_boot(&cs40l25_driver, &boot_state.fw_info);`

> If a system is configured to have separate fw_img blocks for Halo Core™ DSP firmware and tuning, then the process described above must be followed for each fw_img block.

2.4.5 Including and Adding to the Extended Public API

The Extended Public API can be included in the driver build simply by compiling `cs40l25/cs40l25_ext.c` and linking it along with the other driver object files to either a driver library or the rest of the system firmware. It should be noted that the API available in the header `cs40l25/ cs40l25_ext.h` may depend on which firmware algorithms are present in the fw_img data. The available algorithms are listed both as C preprocessor defines in `cs40l25/cs40l25_sym.h` as well as a table in the fw_img_v2 data itself (see "Algorithm ID List" in Appendix A—fw_img_v2 Specification. Additional APIs can easily be added to the Extended Public API available in `cs40l25/ cs40l25_ext.c`.

2.5 Building the Driver

The source files, include paths, and compiler flags required for compiling the CS40L25 MCU driver are described in the table below.

Table 8 Driver Build Requirements

Requirement Type	Requirement	Notes
Source files	cs40l25/cs40l25.c	
	cs40l25/config/cs40l25_syscfg_regs.c	
	common/fw_img.c	
	cs40l25/cs40l25_ext.c	Optional—only required if the Extended Public API is called.
	cs40l25/<path to fw_img>/<fw_img filename>	Optional—only required if the fw_img for Halo Core™ DSP firmware or configuration is stored in the MCU ROM.
Include paths	cs40l25	
	cs40l25/config	
	common	
	cs40l25/<path to fw_img>	Optional—only required if the fw_img for Halo Core™ DSP firmware or configuration is stored in the MCU ROM.
Compiler flags	CONFIG_OPEN_LOOP	Required only when running the CS40L25 in a clockless hardware configuration.
	CONFIG_EXT_BOOST	Compiler flag for BSP code (`hw_0_bsp_cs40l25.c`) to select firmware and system configuration for using external boost.

3 Driver Design

The design of the CS40L25 MCU driver is described below in terms of architecture, interfaces, and implementation.

3.1 Driver Architecture

The MCU Driver Software Stack Block Diagram shows the driver library module in relation to other main components of the entire host software system. As seen below, the primary external interface that the driver module is designed to is the BSP-to-Driver Interface. There are other interfaces to common modules that the CS40L25 driver source code relies on, however those are internally developed and are not necessarily exposed to the system once the CS40L25 driver is compiled in library form. The different shadings are to indicate the most likely source of the software components.

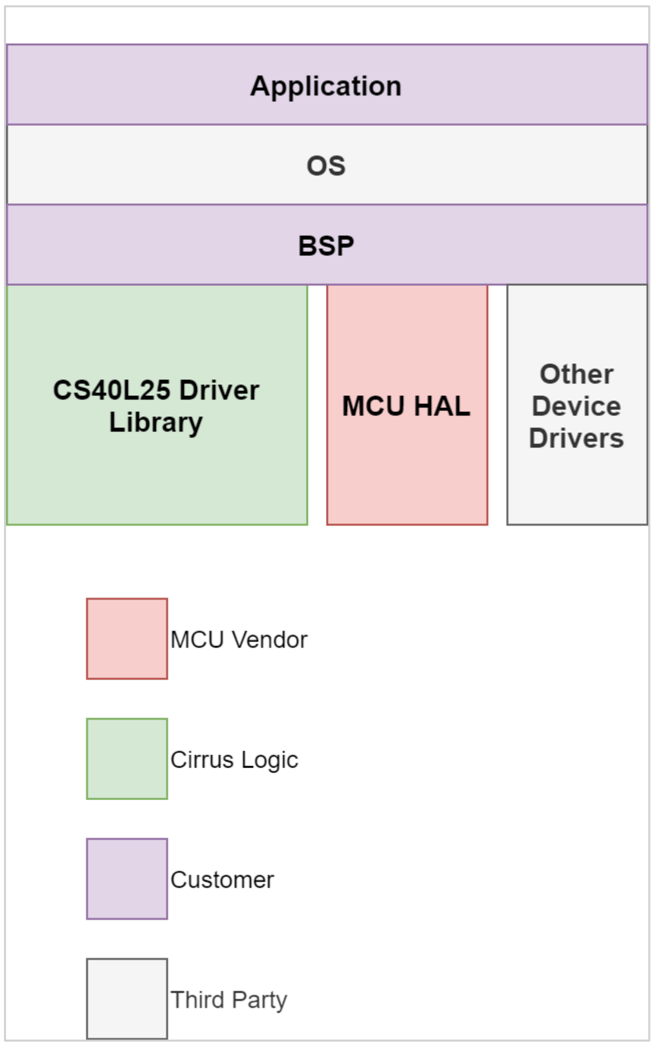

Figure 1 MCU Driver Software Stack

3.1.1 Driver Library Module Description

The CS40L25 MCU driver library will consist of the modules described below.

3.1.1.1 cs40l25.c

The `cs40l25.c` module is the core driver module. Along with the header, this module consists of all the types, defines, functions, and variables required for the basic operation of the CS40L25.

3.1.1.2 cs40l25_syscfg_regs.c

The `cs40l25_syscfg_regs.c` module consists of an array of HW register address/bitfield mask/bitfield value that is delivered to the CS40L25 just before booting completes. This will contain all the hardware-specific configurations the part requires for proper operation. Please note that this module is automatically generated when building the driver library as the output of `wisce_script_converter.py`. For more information on the configuration process to generate the desired `cs40l25_syscfg_regs.c`, please see the integration section Generate Base Configuration.

3.1.1.3 fw_img.c

The `fw_img.c` module is a 'common' module that contains code for parsing the Halo Core™ DSP Firmware and Tuning/Wavetable contents contained in the fw_img_v2 format. fw_img_v2 is a flexible format and parsing of it provides the following benefits:

- Allows Firmware and Tuning/Wavetable contents to be stored either in the MCU non-volatile memory, or in memory external to the MCU such as a serial EEPROM.
- Allows the Driver Library to be independent of a particular firmware release, such that a fw_img_v2 image can be updated without the need to recompile the Driver Library.

For more information on the process of calling the API in this module, please see Processing a fw_img Block.

3.1.1.4 cs40l25_ext.c

The `cs40l25_ext.c` module provides the Driver Library Extended Public API. The functionality provided includes higher-level interfaces for controlling the CS40L25 operation, including algorithm-specific interfaces. Please note that the functionality provided here is optional to the library build and can be excluded to have the minimum library size. Please see Including and Adding to the Extended Public API for more details.

3.1.2 Example Project Modules

The modules described below are not linked in to the CS40L25 MCU driver library but are given as examples in the MCU Driver Software Package to illustrate the library usage and facilitate the integration process.

3.1.2.1 hw_0_bsp.c

The `hw_0_bsp.c` contains the Board Support Package (BSP) code for setting up the development and test hardware described in Example Projects. Of particular importance is that this module implements the BSP-to-Driver interface defined in `/common/bsp_driver_if.h`, the interface which the driver depends on for operating host MCU hardware and registering the IRQ callback.

3.1.2.2 hw_0_bsp_cs40l25.c

The `hw_0_bsp_cs40l25.c` module contains system-level abstractions for the calls to the CS40L25 MCU driver library. This module is the primary module for showing the developer how to call the Driver Public API and Driver Extended Public API. Of particular importance is the sequence of calls required to process a fw_img_v2 image when booting the CS40L25.

3.1.2.3 cs40l25_fw_img.c / cs40l25_cal_fw_img.c

The `cs40l25_fw_img.c` and `cs40l25_cal_fw_img.c` modules consist of the fw_img_v2 image for the CS40L25Haptics Control Firmware release 8.12.2, as well as the default wavetable included with the firmware release WISCE plugin. These modules are automatically generated by calls to the `firmware_converter.py` tool. The command used to generate the files can be found just after the copyright notice in the comments at the top of the file. These files are optional, as the contents of the files could be retrieved in a manner outside the host MCU firmware. For generating new modules, i.e., for new firmware or for an updated wavetable, please see the section Firmware Converter.

3.1.3 Module Dependency

The MCU Driver Dependency Graph shows more specifically the top-level dependencies of the CS40L25 MCU driver source code. Arrows are in the direction of dependence, i.e., the cs40l25.c module is dependent on the hw_0_bsp.c module (which implements the bsp_driver_if.h interface).

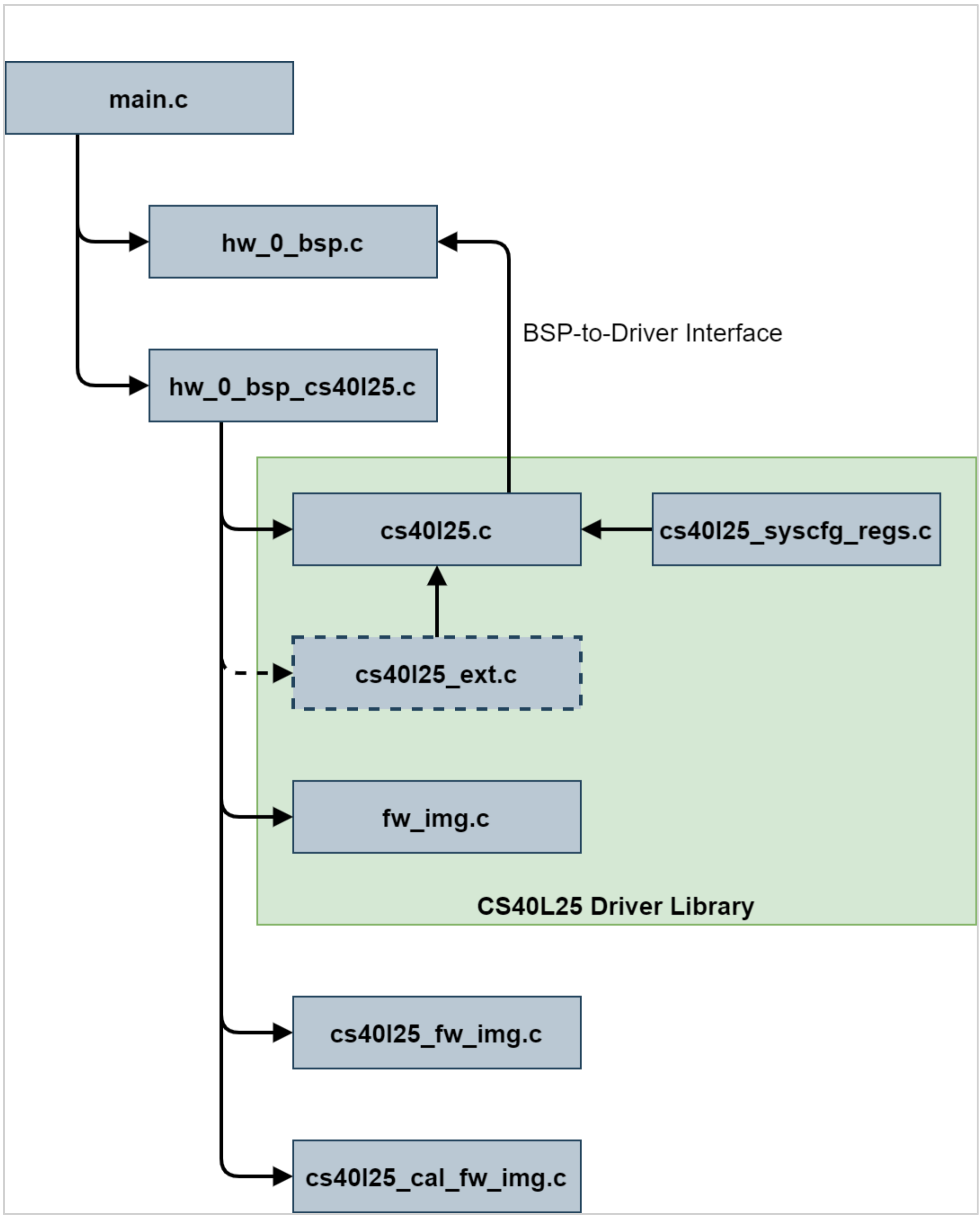

Figure 2 MCU Driver Dependency Graph

3.1.4 Module Components

The MCU Driver Architecture Block Diagram below illustrates the significant components of the CS40L25 MCU driver, the components' interconnections, and the interfaces connections to components in the Board Support Package (BSP). Some aspects that may be helpful to note:

- The driver does not allocate any non-const static variables. Thus, the large data structures for the driver configuration, state, and boot state must be allocated by the Board Support Package (BSP) and passed as parameters to the driver Public API.
- A bsp-to-driver callback is registered by the driver for ALERT IRQ GPI level interrupt event.
- A driver-to-bsp callback is registered by the Board Support Package (BSP) for Notification of Events and Errors.

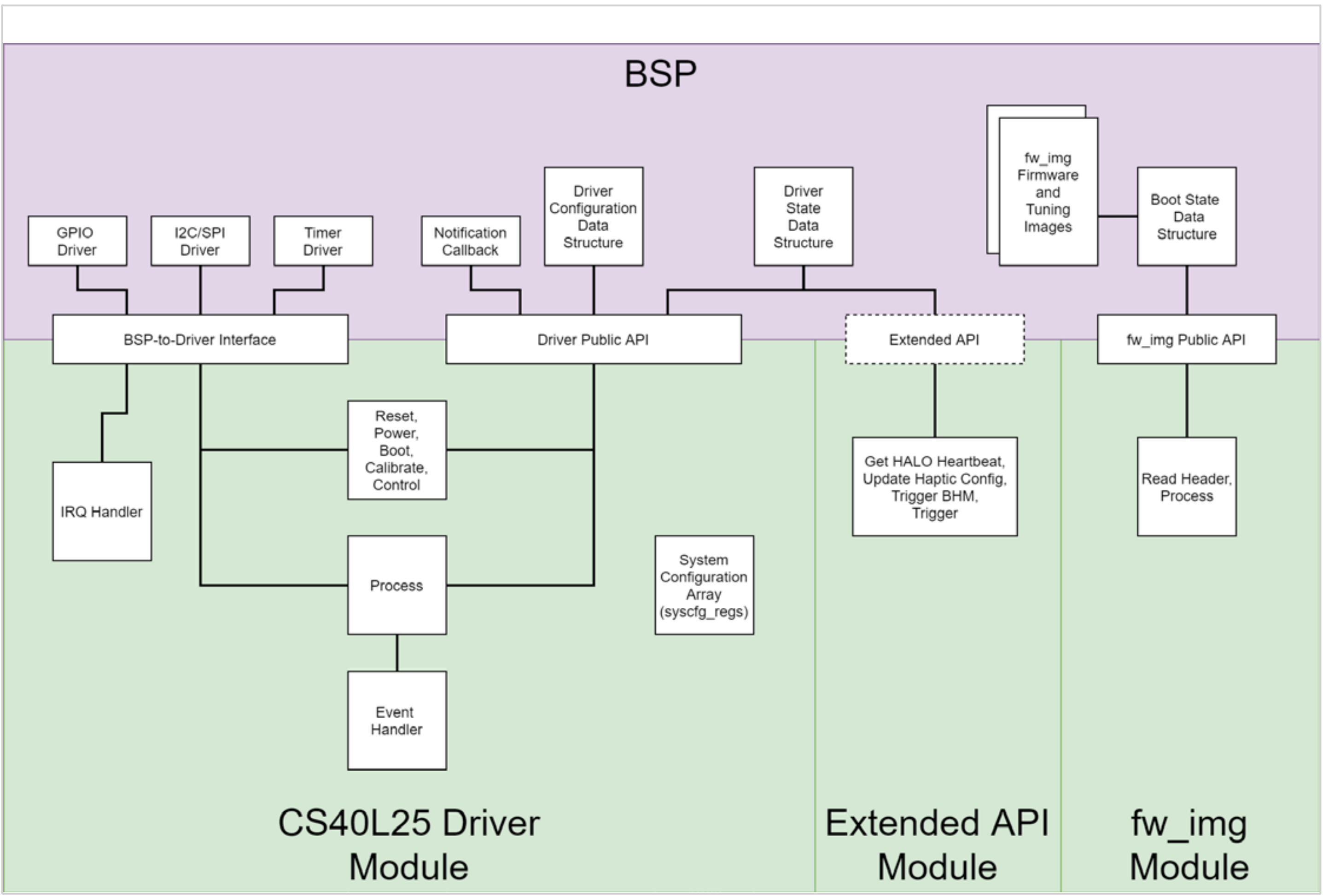

Figure 3 MCU Driver Architecture Block Diagram

A sequence of basic operation of the driver can be summarized as follows:

1. The Board Support Package (BSP) will allocate driver state, configuration, boot state, and firmware and/or coefficient/wav-table files.
2. The Board Support Package (BSP) will then use the Driver Public API to initialize and configure the driver.
3. The driver will use BSP-to-Driver interface to initialize the GPIO level interrupt callback for the CS40L25 INTb pin.
4. The Board Support Package (BSP) will use Driver Public API to request operations like reset, boot, power up, and calibrate.
5. The calls to the Driver Public API block until completed.
6. If at any time the GPI IRQ callback is called, the Event Handler is called next time the process() Driver Public API is called.
7. If a driver event or error has occurred that requires notification of the Board Support Package (BSP), the

Notification callback will be called.
8. Once all driver events have been handled and notified, process() will return and other Driver Public API calls can occur.

3.2 Driver Interfaces

Integration of the CS40L25 MCU driver needs to be concerned with only two interfaces: the Driver Public API, and the BSP-to-Driver Interface. The latter is implemented as a **struct** of function pointers. Details of these interfaces are given in the following sections.

3.2.1 Driver Public API

The CS40L25 MCU driver Public API is defined in `cs40l25/cs40l25.h`.

Public API Definition

```
uint32_t cs40l25_initialize(cs40l25_t *driver);
uint32_t cs40l25_configure(cs40l25_t *driver, cs40l25_config_t *config);
uint32_t cs40l25_process(cs40l25_t *driver);
uint32_t cs40l25_control(cs40l25_t *driver, cs40l25_control_request_t req);
uint32_t cs40l25_reset(cs40l25_t *driver);
uint32_t cs40l25_write_block(cs40l25_t *driver, uint32_t addr, uint8_t *data, uint32_t size);
uint32_t cs40l25_boot(cs40l25_t *driver, fw_img_info_t *fw_info);
uint32_t cs40l25_power(cs40l25_t *driver, uint32_t power_state);
uint32_t cs40l25_calibrate(cs40l25_t *driver, uint32_t calib_type);
uint32_t cs40l25_start_i2s(cs40l25_t *driver);
uint32_t cs40l25_stop_i2s(cs40l25_t *driver);
uint32_t cs40l25_enable_vamp_discharge(cs40l25_t *driver, bool is_enable);
```

A few important notes are listed below:

- Each API requires a parameter of a driver handle (pointer to driver state), meaning this API contains no static state and can be easily used for systems containing multiple CS40L25devices
- Various types of variables must be allocated by calling code in order to operate the public API:
  – `cs40l25_t`
  – `cs40l25_config_t`
  – `fw_img_info_t`
- These types also contain components of the following types to note:
  – `cs40l25_notification_callback_t`

Each Driver Public API returns a constant of the set below:

Public API Return Codes

```
#define CS40L25_STATUS_OK                                    (0)
#define CS40L25_STATUS_FAIL                                  (1)
```

The Driver Public API functions are described in tables in the following sections. The documentation here is imported directly from the Doxygen-generated documentation for the Public API source code.

3.2.1.1 cs40l25_initialize

Table 9 cs40l25_initialize API Description

Definition	uint32_t cs40l25_initialize (cs40l25_t *driver)
Brief	Initialize driver state/handle.
Details	Sets all driver state members to 0.
Attention/Warning	
Parameters	[in] `driver` - Pointer to the driver state
Returns	• CS40L25_STATUS_FAIL if driver is NULL • CS40L25_STATUS_OK otherwise

3.2.1.2 cs40l25_configure

Table 10 cs40l25_configure API Description

Definition	uint32_t cs40l25_configure (cs40l25_t *driver, cs40l25_config_t *config)
Brief	Configures driver state/handle.
Details	Including the following: • Applies all one-time configurations to the driver state • Registers the IRQ Callback for INTb GPIO with the Board Support Package (BSP) • Applies calibration data (if valid) to the driver state
Attention/Warning	
Parameters	[in] `driver` - Pointer to the driver state [in] `config` - Pointer to driver configuration data structure
Returns	• `CS40L25_STATUS_FAIL` if any pointers are NULL • `CS40L25_STATUS_OK` otherwise

The members of the `cs40l25_config_t` configuration data structure are described below:

- `bsp_config`–Parameters used for calling the `bsp_driver_if_g` interface
- syscfg_regs–Pointer to the system configuration register settings (see Generate Base Configuration)
- syscfg_regs_total–Length of the system configuration register settings
- cal_data–Calibration parameters as a result of running a calibration sequence
- event_control–Settings the Halo Core™ DSP Firmware that result in generating events that trigger the ALERTb pin
- ext_boost–Settings for enabling and configuring the driver for External Boost Mode

3.2.1.3 cs40l25_process

Table 11 cs40l25_process API Description

Definition	uint32_t cs40l25_process (cs40l25_t *driver)
Brief	Processes driver events and errors
Details	If cs40l25_irq_callback has been called since the last time process() was called: • processes the event handling routine. • calling Board Support Package (BSP) callbacks if relevant events detected. Returns immediately if cs40l25_irq_callback hasn't been called.
Attention/Warning	This MUST be placed either in baremetal or RTOS task while (1)
Parameters	[in] `driver`–Pointer to the driver state
Returns	• Returns CS40L25_STATUS_OK. • Any errors will result in the driver state being set to CS40L25_STATE_ERROR.

3.2.1.4 cs40l25_control

Table 12 cs40l25_control API Description

Definition	uint32_t cs40l25_control (cs40l25_t *driver, cs40l25_control_request_t req)
Brief	Submit a Control Request to the driver.
Details	Caller will initialize a cs40l25_control_request_t 'req' based on the control it wishes to access. This request will then be processed and return once the requested actions have completed.
Attention/Warning	
Parameters	[in] `driver`–Pointer to the driver state [in] `req`–data structure for control request passed by value
Returns	• CS40L25_STATUS_FAIL if Control Request ID is invalid OR if executing the Control Request fails • CS40L25_STATUS_OK otherwise

3.2.1.5 cs40l25_reset

Table 13 cs40l25_reset API Description

Definition	uint32_t cs40l25_reset (cs40l25_t *driver);
Brief	Reset the CS40L25 in Basic Haptics Mode (BHM)
Details	Reset will take the following actions: • Toggle RESET GPIO. • Identify the device. • Ensure BHM has booted successfully.
Attention/Warning	
Parameters	[in] `driver` - Pointer to the driver state.
Returns	• CS40L25_STATUS_FAIL if any part of the device bring up fails. • CS40L25_STATUS_OK otherwise.

3.2.1.6 cs40l25_write_block

Table 14 cs40l25_write_block API Description

Definition	uint32_t cs40l25_write_block (cs40l25_t *driver, uint32_t addr, uint8_t *data, uint32_t size)
Brief	Write data to the CS40L25 Halo Core™ DSP memory
Details	Writes the data pointed to by *data*, of *size* bytes, to the address at *addr*. Typically this is used when decoding a fw_img data block.
Attention/Warning	
Parameters	[in] `driver` - Pointer to the driver state. [in] `addr`–The 32-bit absolute address in the Halo Core™ DSP memory that the data should be written. [in] `data`–A pointer to the data to be written out to the Halo Core™ DSP memory. [in] `size`–The size of the data pointed to by `data`.
Returns	• CS40L25_STATUS_FAIL if any of the inputs are invalid or the write to Halo Core™ DSP memory fails. • CS40L25_STATUS_OK otherwise.

3.2.1.7 cs40l25_boot

Table 15 cs40l25_boot API Description

Definition	uint32_t cs40l25_boot (cs40l25_t *driver, fw_img_info_t *fw_info)
Brief	Boot the CS40L25
Details	If `fw_info` is NULL, this will signal to the driver that any previously loaded firmware, is no longer valid. If `fw_info` is not NULL, boot will: • Populate the driver's internal fw_info reference to the struct passed in. • Initialize the populate the wseq table. • Apply any post-boot config. • If the firmware isn't a calibration firmware:

Definition	uint32_t cs40l25_boot (cs40l25_t *driver, fw_img_info_t *fw_info)
	o Apply any stored calibration data. • Write out any configuration generated by syscfg. • Write out the IRQ mask sequence.
Attention/Warning	
Parameters	[in] `driver`—Pointer to the driver state. [in] `fw_info`—Pointer to a valid `fw_img_info_t` struct.
Returns	• CS40L25_STATUS_FAIL if any firmware symbols cannot be found in the loaded firmware's symbol table. • CS40L25_STATUS_OK otherwise.

3.2.1.8 cs40l25_power

Table 16 cs40l25_power API Description

Definition	uint32_t cs40l25_power (cs40l25_t *driver, uint32_t power_state)
Brief	Change the power state
Details	This submits a request to Power Up, Power Down, Wake or Hibernate the CS40L25.
Attention/Warning	
Parameters	[in] `driver`—Pointer to the driver state [in] `power_state`—New power state
Returns	• CS40L25_STATUS_FAIL if power state transition requested isn't valid. • CS40L25_STATUS_OK otherwise.

The possible choices for `power_state` are listed below:

- `CS40L25_POWER_UP`
- `CS40L25_POWER_DOWN`
- `CS40L25_POWER_HIBERNATE`
- `CS40L25_POWER_WAKE`

3.2.1.9 cs40l25_calibrate

Table 17 cs40l25_calibrate API Description

Definition	uint32_t cs40l25_calibrate (cs40l25_t *driver, uint32_t calib_type)
Brief	Calibrate the Halo Core™ DSP firmware.
Details	This performs the calibration procedure required for the CS40L25 Haptic Control firmware to determine the resonant frequency (F0), DC resistance (ReDC), and Q-factor (Q) of an actuator.
Attention/Warning	
Parameters	[in] `driver`—Pointer to the driver state [in] `calib_type`—Calibration type to be executed. Valid options can be found in `cs40l25.h`, under the `CS40L25_CALIB_` defines.
Returns	CS40L25_STATUS_FAIL if the calibration procedure fails. CS40L25_STATUS_OK otherwise.

The possible options for `calib_type` are listed below:

- `CS40L25_CALIB_F0`
- `CS40L25_CALIB_QEST`
- `CS40L25_CALIB_ALL`

After calibration is complete, the calibration parameters are saved to the driver state structure at `cs40l25_t.config.cal_data`. The host software can then save these parameters to non-volatile memory for application upon the next reset.

3.2.1.10 cs40l25_start_i2s

Table 18 cs40l25_start_i2s API Description

Definition	uint32_t cs40l25_start_i2s (cs40l25_t *driver)
Brief	Put the Halo Core™ DSP into I2S streaming mode.
Details	Performs all register/memory field address updates required to put the Halo Core™ DSP in to I2S Streaming Mode, including moving the pll ref src to the sclk.
Attention/Warning	
Parameters	[in] `driver`—Pointer to the driver state
Returns	• CS40L25_STATUS_FAIL if any of the steps to enable I2S fail • CS40L25_STATUS_OK otherwise.

3.2.1.11 cs40l25_stop_i2s

Table 19 cs40l25_stop_i2s API Description

Definition	uint32_t cs40l25_stop_i2s (cs40l25_t *driver)
Brief	Take the Halo Core™ DSP back out of I2S streaming mode.
Details	Performs all register/memory field address updates required to take the Halo Core™ DSP back out of I2S Streaming Mode, including moving the pll ref src back to its original source.
Attention/Warning	
Parameters	[in] `driver`—Pointer to the driver state
Returns	• CS40L25_STATUS_FAIL if any of the steps to disable I2S fail • CS40L25_STATUS_OK otherwise.

3.2.1.12 cs40l25_enable_vamp_discharge

Table 20 cs40l25_enable_vamp_discharge API Description

Definition	uint32_t cs40l25_enable_vamp_discharge (cs40l25_t *driver, bool is_enable)
Brief	Enable discharging of VAMP when in External Boost mode
Details	Writes to the Halo Core™ DSP firmware control to enable/disable discharging of the VAMP supply. For example usage, see the call sequence for `bsp_dut_discharge_vamp()` called in `cs40l25/baremetal/main_ext_boost.c`
Attention/Warning	
Parameters	[in] `driver`—Pointer to the driver state [in] `is_enable`—'true' for enable VAMP discharge, 'false' for disable VAMP discharge
Returns	• CS40L25_STATUS_FAIL if driver is not in External Boost Mode, Control port activity fails, or MBOX write ACK fails • CS40L25_STATUS_OK otherwise.

3.2.2 Driver Extended Public API

The CS40L25 MCU driver Extended Public API is defined in `cs40l25/cs40l25_ext.h`.

Public API Definition

```
uint32_t cs40l25_get_halo_heartbeat(cs40l25_t *driver, uint32_t *hb);
uint32_t cs40l25_update_haptic_config(cs40l25_t *driver, cs40l25_haptic_config_t *config);
uint32_t cs40l25_trigger_bhm(cs40l25_t *driver);
uint32_t cs40l25_trigger(cs40l25_t *driver, uint32_t index, uint32_t duration_ms);
uint32_t cs40l25_set_click_compensation_enable(cs40l25_t *driver, bool enable);
uint32_t cs40l25_set_clab_enable(cs40l25_t *driver, bool enable);
uint32_t cs40l25_set_clab_peak_amplitude(cs40l25_t *driver, uint32_t amplitude);
uint32_t cs40l25_set_dynamic_f0_enable(cs40l25_t *driver, bool enable);
uint32_t cs40l25_get_dynamic_f0(cs40l25_t *driver, cs40l25_dynamic_f0_table_entry_t *f0_entry);
uint32_t cs40l25_get_dynamic_redc(cs40l25_t *driver, uint32_t *redc);
```

The Extended API calls share the same constant return codes listed for the Driver Public API.

The Driver Extended Public API functions are described in tables in the following sections. The documentation here is imported directly from the Doxygen-generated documentation for the Extended Public API source code. Please note if any algorithms are required to be present in the Halo Core™ DSP firmware for the API to valid.

3.2.2.1 cs40l25_get_halo_heartbeat

Table 21 cs40l25_get_halo_heartbeat API Description

Definition	uint32_t cs40l25_get_halo_heartbeat (cs40l25_t *driver, uint32_t *hb)
Brief	Get the HALO HEARTBEAT
Required Algorithm	None
Details	Get the current value of the FW control HALO HEARTBEAT. If running in ROM mode (BHM), the ROM HALO HEARTBEAT will be returned. If running in RAM mode, the loaded FW's HALO HEARTBEAT will be returned.
Attention/Warning	
Parameters	[in] `driver`— Pointer to the driver state. [in/out] `hb`—Pointer to heartbeat count
Returns	• `CS40L25_STATUS_FAIL` if get of heartbeat failed, or if hb is NULL • `CS40L25_STATUS_OK` otherwise

3.2.2.2 cs40l25_update_haptic_config

Table 22 cs40l25_update_haptic_config API Description

Definition	uint32_t cs40l25_update_haptic_config (cs40l25_t *driver, cs40l25_haptic_config_t *config)
Brief	Update all the required Halo Core™ DSP FW controls to set up for the specific haptic configuration.
Required Algorithm	None
Details	Update all the required Halo Core™ DSP FW controls to set up for the specific haptic configuration. See `cs40l25_haptic_config_t` in `/cs40l25/cs40l25_ext.h` for more details.
Attention/Warning	
Parameters	[in] `driver`—Pointer to the driver state. [in] `config`—Pointer to haptic configuration to use for update
Returns	• `CS40L25_STATUS_FAIL` if update of any Halo Core™ DSP FW control fails, or if config is NULL • `CS40L25_STATUS_OK` otherwise

3.2.2.3 cs40l25_trigger_bhm

Table 23 cs40l25_trigger_bhm API Description

Definition	uint32_t cs40l25_trigger_bhm (cs40l25_t *driver)
Brief	Trigger the ROM Mode (BHM) Haptic Effect
Required Algorithm	None
Details	
Attention/Warning	This call will write to the required ROM Mode FW Control whether or not the L25 is currently in either ROM or RAM modes. If in RAM mode, the user should expect no effect from calls to this function.
Parameters	[in] `driver`—Pointer to the driver state
Returns	• `CS40L25_STATUS_FAIL` if update of any Halo Core™ DSP FW control fails • `CS40L25_STATUS_OK` otherwise

3.2.2.4 cs40l25_trigger

Table 24 cs40l25_trigger API Description

Definition	uint32_t cs40l25_trigger (cs40l25_t *driver, uint32_t index, uint32_t duration_ms)
Brief	Trigger RAM Mode Haptic Effects
Required Algorithm	VIBEGEN
Details	
Attention/Warning	
Parameters	[in] `driver`—Pointer to the driver state [in] `index`—Index into the Halo Core™ DSP FW Wavetable [in] `duration_ms`—Duration of effect playback in milliseconds
Returns	• `CS40L25_STATUS_FAIL` if update of any Halo Core™ DSP FW control fails • `CS40L25_STATUS_OK` otherwise

3.2.2.5 cs40l25_set_click_compensation_enable

Table 25 cs40l25_set_click_compensation_enable API Description

Definition	uint32_t cs40l25_set_click_compensation_enable (cs40l25_t *driver, bool enable)
Brief	Enable the Halo Core™ DSP FW Click Compensation
Required Algorithm	VIBEGEN
Details	
Attention/Warning	
Parameters	[in] `driver`—Pointer to the driver state [in] `enable`—true to enable Click Compensation, false to disable Click Compensation
Returns	• `CS40L25_STATUS_FAIL` if update of any Halo Core™ DSP FW control fails • `CS40L25_STATUS_OK` otherwise

3.2.2.6 cs40l25_set_clab_enable

Table 26 cs40l25_set_clab_enable API Description

Definition	uint32_t cs40l25_set_clab_enable (cs40l25_t *driver, bool enable)
Brief	Enable the Halo Core™ DSP FW CLAB Algorithm
Required Algorithm	CLAB
Details	
Attention/Warning	
Parameters	[in] `driver`—Pointer to the driver state [in] `enable`—true to enable CLAB, false to disable CLAB
Returns	• `CS40L25_STATUS_FAIL` if update of any Halo Core™ DSP FW control fails

Definition	uint32_t cs40l25_set_clab_enable (cs40l25_t *driver, bool enable)
	• CS40L25_STATUS_OK otherwise

3.2.2.7 cs40l25_set_clab_peak_amplitude

Table 27 cs40l25_set_clab_peak_amplitude API Description

Definition	uint32_t cs40l25_set_clab_peak_amplitude (cs40l25_t *driver, uint32_t amplitude)
Brief	Set the CLAB Peak Amplitude Control
Required Algorithm	CLAB
Details	
Attention/Warning	
Parameters	[in] driver—Pointer to the driver state [in] amplitude—setting for Peak Amplitude Control
Returns	• CS40L25_STATUS_FAIL if update of any Halo Core™ DSP FW control fails • CS40L25_STATUS_OK otherwise

3.2.2.8 cs40l25_set_dynamic_f0_enable

Table 28 cs40l25_set_dynamic_f0_enable API Description

Definition	uint32_t cs40l25_set_dynamic_f0_enable (cs40l25_t *driver, bool enable)
Brief	Enable the Halo Core™ DSP FW Dynamic F0 Algorithm
Required Algorithm	DYNAMIC_F0
Details	
Attention/Warning	
Parameters	[in] driver—Pointer to the driver state [in] enable— true to enable Dynamic F0, false to disable Dynamic F0
Returns	• CS40L25_STATUS_FAIL if update of any Halo Core™ DSP FW control fails • CS40L25_STATUS_OK otherwise

3.2.2.9 cs40l25_get_dynamic_f0

Table 29 cs40l25_get_dynamic_f0 API Description

Definition	uint32_t cs40l25_get_dynamic_f0(cs40l25_t *driver, cs40l25_dynamic_f0_table_entry_t *f0_entry)
Brief	Get the Dynamic F0
Required Algorithm	DYNAMIC_F0
Details	Get the current value of the F0 for a specific index into the WaveTable. The index is specified in the f0_entry member index. The current F0 for WaveTable entries are stored in a Dynamic F0 table in FW, which only contains a Dynamic F0 for WaveTable entries that have been played since power up. This table has a maximum size of 20. If the index specified is not found in the FW table, the table default CS40L25_DYNAMIC_F0_TABLE_ENTRY_DEFAULT is returned.
Attention/Warning	
Parameters	[in] driver—Pointer to the driver state [in/out] f0_entry— Pointer to Dynamic F0 structure. See cs40l25_dynamic_f0_table_entry_t in /cs40l25/cs40l25_ext.h for more details.
Returns	• CS40L25_STATUS_FAIL if update of any Halo Core™ DSP FW control fails; if the specified WaveTable index is ≥ 20 • CS40L25_STATUS_OK otherwise

3.2.2.10 cs40l25_get_dynamic_redc

Table 30 cs40l25_get_dynamic_redc API Description

Definition	uint32_t cs40l25_get_dynamic_redc (cs40l25_t *driver, uint32_t *redc)
Brief	Get the Dynamic ReDC
Required Algorithm	DYNAMIC_F0
Details	Get the current value of the Dynamic ReDC for the attached actuator. If an invalid value is read, the driver will wait 10 milliseconds before reading again. It will attempt 30 reads of ReDC before failing.
Attention/Warning	
Parameters	[in] `driver`–Pointer to the driver state [out] `redc`–Pointer to Dynamic ReDC value
Returns	• `CS40L25_STATUS_FAIL` if update of any Halo Core™ DSP FW control fails; if the specified WaveTable index is ≥ 20 • `CS40L25_STATUS_OK` otherwise

3.2.3 fw_img Public API

The CS40L25 MCU driver fw_img Public API is defined in `common/fw_img.h`.

fw_img Public API Definition

```
extern uint32_t fw_img_read_header(fw_img_boot_state_t *state);
extern uint32_t fw_img_process(fw_img_boot_state_t *state);
```

Each Driver Public API returns a constant of the set below:

fw_img Public API Return Codes

```
#define FW_IMG_STATUS_OK                                   (0)
#define FW_IMG_STATUS_FAIL                                 (1)
#define FW_IMG_STATUS_AGAIN                                (2)
#define FW_IMG_STATUS_NODATA                               (4)
#define FW_IMG_STATUS_DATA_READY                           (5)
```

The fw_img Public API functions are described in tables in the following sections. The documentation here is imported directly from the Doxygen-generated documentation for the fw_img Public API source code.

3.2.3.1 fw_img_read_header

Table 31 fw_img_read_header API Description

Definition	uint32_t fw_img_read_header(fw_img_boot_state_t *state)
Brief	Read fw_img header
Details	Reads all members into fw_img_boot_state_t member fw_info.header
Attention/Warning	
Parameters	[in] `state` - Pointer to the fw_img boot state
Returns	• FW_IMG_STATUS_FAIL if: o any NULL pointers o fw_img_blocks_size is 0header magic number is incorrect • FW_IMG_STATUS_OK otherwise

3.2.3.2 fw_img_process

Table 32 fw_img_process API Description

Definition	**uint32_t fw_img_process(fw_img_boot_state_t *state)**
Brief	Process more fw_img bytes
Details	Continues processing fw_img bytes and updating the fw_img_boot_state_t according to the state machine.
Attention/Warning	
Parameters	[in] `state` - Pointer to the fw_img boot state
Returns	• FW_IMG_STATUS_FAIL if: o any NULL pointers o any errors processing fw_img data • FW_IMG_STATUS_NODATA if fw_img_process() requires input of another block of fw_img data • FW_IMG_STATUS_DATA_READY if an output block of data is ready to be sent to the device • FW_IMG_STATUS_OK Once finished reading the fw_img footer

3.2.4 BSP-to-Driver Interface

The BSP-to-Driver Interface to which the CS40L25 MCU driver is designed is defined in the type `bsp_driver_if_t`, which defines a struct of function pointers—one for each API. The CS40L25 MCU driver accesses the BSP-to-Driver Interface via the instance pointer for this definition - `bsp_driver_if_g`. However, it is up to the system designer to implement the API below, including instantiation of `bsp_driver_if_g`. The BSP-to-Driver Interface is defined in `/common/bsp_driver_if.h`.

bsp_driver_if_t Definition and Instance

```
typedef struct
{
    uint32_t (*set_gpio)(uint32_t gpio_id, uint8_t gpio_state);
    uint32_t (*set_supply)(uint32_t supply_id, uint8_t supply_state);
    uint32_t (*register_gpio_cb)(uint32_t gpio_id, bsp_callback_t cb, void *cb_arg);
    uint32_t (*set_timer)(uint32_t duration_ms, bsp_callback_t cb, void *cb_arg);
    uint32_t (*i2c_reset)(uint32_t bsp_dev_id, bool *was_i2c_busy);
    uint32_t (*i2c_read_repeated_start)(uint32_t bsp_dev_id,
                                        uint8_t *write_buffer,
                                        uint32_t write_length,
                                        uint8_t *read_buffer,
                                        uint32_t read_length,
                                        bsp_callback_t cb,
                                        void *cb_arg);
    uint32_t (*i2c_write)(uint32_t bsp_dev_id,
                          uint8_t *write_buffer,
                          uint32_t write_length,
                          bsp_callback_t cb,
                          void *cb_arg);
    uint32_t (*i2c_db_write)(uint32_t bsp_dev_id,
                          uint8_t *write_buffer_0,
                          uint32_t write_length_0,
                          uint8_t *write_buffer_1,
                          uint32_t write_length_1,
                          bsp_callback_t cb,
                          void *cb_arg);
    uint32_t (*spi_read)(uint32_t bsp_dev_id,
                         uint8_t *addr_buffer,
                         uint32_t addr_length,
                         uint8_t *data_buffer,
                         uint32_t data_length,
                         uint32_t pad_len);
    uint32_t (*spi_write)(uint32_t bsp_dev_id,
                          uint8_t *addr_buffer,
                          uint32_t addr_length,
                          uint8_t *data_buffer,
                          uint32_t data_length,
                          uint32_t pad_len);
    uint32_t (*enable_irq)(void);
    uint32_t (*disable_irq)(void);
    uint32_t (*spi_throttle_speed)(uint32_t speed_hz);
    uint32_t (*spi_restore_speed)(void);
} bsp_driver_if_t;
extern bsp_driver_if_t *bsp_driver_if_g;
```

The BSP-to-Driver API functions return a constant of the set below:

BSP-to-Driver Interface Return Codes

```
#define BSP_STATUS_OK                  (0)
#define BSP_STATUS_FAIL                (1)
```

The BSP-to-Driver API functions required for the CS40L25 MCU driver are summarized in the table below. If a function declared in `bsp_driver_if_t` is listed as 'Not required', it is entry in the definition of the function pointer table can be set to an error handler. Further documentation can be found in the Doxygen-generated documentation found in the package file-system at `/cs40l25/doc.zip`.

Table 33 BSP-to-Driver Interface Description

Name	Brief
set_gpio()	Set GPIO to LOW/HIGH.
set_supply()	Not required
register_gpio_cb()	Register GPIO level callback. Used for registering the driver API to call when the CS40L25's interrupt is low.
set_timer()	Set a timer to expire.
i2c_reset()	Not required
i2c_read_repeated_start()	Perform an I2C Write-Repeated Start-Read transaction.
i2c_write()	Perform I2C Write.
i2c_db_write()	Perform a Double-Buffered ("db") I2C Write.
spi_read()	Perform SPI read transaction.
spi_write()	Perform SPI write transaction.
enable_irq()	Global enable of interrupts.
disable_irq()	Global disable of interrupts.
spi_throttle_speed()	Not required
spi_restore_speed()	Not required

3.3 Driver Implementation

3.3.1 Driver Modes and States

The API implementation operates on the following primary parameters of the driver handle/state `cs40l25_t`:

- driver state (`cs40l25_t.state`)
- driver mode (`cs40l25_t.mode`)

The two modes the driver can be in are:

- `CS40L25_MODE_HANDLING_CONTROLS`–The driver is currently able to process control requests.
- `CS40L25_MODE_HANDLING_EVENTS`–The driver has received an IRQ and is not able to process control requests, but will process pending events with the next call of cs40l25_process().

The driver states and valid public API are summarized in the table below. Please note the following:

- In any state in which 'IRQ Events Processed?' is true, API calls other than cs40l25_process() will not be valid unless the driver mode is `CS40L25_MODE_HANDLING_CONTROLS`.
- In any state in which `cs40l25_control()` is a Valid Public API, then all Extended Public APIs are also valid.
- `cs40l25_reset()` is valid for all states except for: UNCONFIGURED, ERROR.

Table 34 Driver State Description

State	CS40L25 Hardware State	Valid Public API	IRQ Events Processed?	Notes
UNCONFIGURED	unknown	cs40l25_initialize() cs40l25_configure()		
CONFIGURED	unknown	cs40l25_reset()	X	
POWER_UP	Basic Haptics Mode (BHM)	cs40l25_power() cs40l25_control()		Valid power_state value is CS40L25_POWER_DOWN.
STANDBY	Standby	cs40l25_write_block() cs40l25_boot() cs40l25_power() cs40l25_control()		Valid power_state value is CS40L25_POWER_UP.
DSP_POWER_UP	Power Up	cs40l25_power() cs40l25_start_i2s() cs40l25_stop_i2s() cs40l25_control()	X	Valid power_state values are: CS40L25_POWER_DOWN CS40L25_POWER_HIBERNATE
DSP_STANDBY	Standby	cs40l25_power() cs40l25_control()	X	Valid power_state value is CS40L25_POWER_UP.
CAL_POWER_UP	Power Up	cs40l25_calibrate() cs40l25_power()		Valid power_state value is CS40L25_POWER_DOWN.
CAL_STANDBY	Standby	cs40l25_power()		Valid power_state value is CS40L25_POWER_UP.
HIBERNATE	Hibernation	cs40l25_power()	X	IRQ Events will automatically bring the part out of Hibernation. Valid power_state value is CS40L25_POWER_WAKE.
ERROR	unknown	cs40l25_initialize()		

3.3.2 Event Handling

Events that occur in CS40L25 hardware or Halo Core™ DSP firmware are indicated by the active-low ALERTb signal. All event handling for the CS40L25 is handled by the Event Control algorithm in Halo Core™ DSP firmware. Via the BSP-to-Driver Interface API `register_gpio_cb`, during `cs40l25_configure` the driver will have registered a callback for ALERTb events. When an event occurs and the ALERTb callback is called, the driver switches to `CS40L25_MODE_HANDLING_EVENTS` mode. When the driver is in `CS40L25_MODE_HANDLING_EVENTS` mode, the driver completes processing of the current Control Request (if any), and then waits for the `cs40l25_process` Driver Public API to be called. A single call to `cs40l25_process` will result in all events being serviced. The main tasks are:

1. Read each EVENT Halo Core™ DSP Firmware Control
2. Save and clear the Event Flags present in the EVENT Controls. Some Hardware Events required additional steps to be cleared.
3. Notify the Board Support Package (BSP) of the driver Events via the Notification callback (`cs40l25/cs40l25.h` : `cs40l25_config_t.notification_cb`)

Events are enabled by setting corresponding bit-fields in the `event_control` member of `cs40l25_config_t` that is provided when the `cs40l25_configure` Driver Public API is called. Events are reported via bit-fields in the `event_flags` argument given to the BSP Notification callback `cs40l25_config_t.notification_cb`.

The table below lists all the possible events that can be reported via the CS40L25 MCU driver along with the bit-field required to enable the event.

Table 35 Event Handler Bit-Field Table

Event	event_control Bit Field	event_flags Bit Field
GPIO1 pressed	CS40L25_EVENT_FLAG_GPIO_1_PRESS	gpio1
GPIO1 released	CS40L25_EVENT_FLAG_GPIO_1_RELEASE	
GPIO2 pressed	CS40L25_EVENT_FLAG_GPIO_2_PRESS	gpio2
GPIO2 released	CS40L25_EVENT_FLAG_GPIO_2_RELEASE	
GPIO3 pressed	CS40L25_EVENT_FLAG_GPIO_3_PRESS	gpio3
GPIO3 released	CS40L25_EVENT_FLAG_GPIO_3_RELEASE	
GPIO4 pressed	CS40L25_EVENT_FLAG_GPIO_4_PRESS	gpio4
GPIO4 released	CS40L25_EVENT_FLAG_GPIO_4_RELEASE	
GPIO-triggered Playback Done	CS40L25_EVENT_FLAG_GPIO_PLAYBACK_DONE	gpio1, gpio2, gpio3, or gpio4
Control Port-triggered playback resumed	CS40L25_EVENT_FLAG_CP_PLAYBACK_RESUME	playback_resume
Control Port-triggered playback suspended	CS40L25_EVENT_FLAG_CP_PLAYBACK_SUSPEND	playback_end_suspend
Control Port-triggered playback done	CS40L25_EVENT_FLAG_CP_PLAYBACK_DONE	
Ready for Data	CS40L25_EVENT_FLAG_READY_FOR_DATA	rx_ready
Boost overvoltage	CS40L25_EVENT_FLAG_BOOST_OVERVOLTAGE	hardware
Boost undervoltage	CS40L25_EVENT_FLAG_BOOST_UNDERVOLTAGE	
Boost inductor short	CS40L25_EVENT_FLAG_BOOST_INDUCTOR_SHORT	
Over-temperature warning	CS40L25_EVENT_FLAG_OVERTEMP_WARNING	
Over-temperature error	CS40L25_EVENT_FLAG_OVERTEMP_ERROR	
Amplifier short	CS40L25_EVENT_FLAG_AMP_SHORT	
Driver state changed to ERROR	CS40L25_EVENT_FLAG_STATE_ERROR	(always enabled)
DSP firmware error	CS40L25_EVENT_FLAG_DSP_ERROR	(always enabled)

4 Tooling

Various tools are provided to aid customer integration of Cirrus Logic parts. The following sections describe them and how they should be used.

4.1 WISCE™ Script Converter

`wisce_script_converter.py` is a Python 3.x tool that allows the creation of initial system configuration register write sequences - it is a Python code generation script that is run by the makefile before building the driver source code.

wisce_script_converter.py takes as an input a WISCE™ profile script that includes the control port writes required to configure the CS40L25. The script generates an array of address/mask/value entries in /cs40l25/cs40l25_syscfg_regs.h/.c.

When using the makefile included in the MCU Driver Software Package, there should be no need for the developer to directly call wisce_script_converter.py. More information on the wisce_script_converter.py usage and building the driver with its output can be found in Generate Base Configuration.

Usage for wisce_script_converter

```
$ python3 tools/wisce_script_converter/wisce_script_converter.py -h

wisce_to_syscfg_reg_converter
Convert from WISCE Script Text file to Alt-OS Syscfg Reg
Version 1.0.0
usage: wisce_script_converter.py [-h] -c {c_array} -p PART -i INPUT
                                 [-o OUTPUT] [--include-comments]

Parse command line arguments

optional arguments:
  -h, --help            show this help message and exit
  -c {c_array}, --command {c_array}
                        The command you wish to execute.
  -p PART, --part PART  The part number text for output.
  -i INPUT, --input INPUT
                        The filename of the WISCE script to be parsed.
  -o OUTPUT, --output OUTPUT
                        The output filename.
  --include-comments    Include comments from the WISCE script.
```

4.2 Firmware Converter

`firmware_converter.py` is a Python 3.x tool that is used to incorporate new CS40L25 Halo Core™ DSP firmware images and/or new tuning/coefficient files into the MCU Driver Software Package.

Currently, the supported firmware_converter.py method for integrating new images into the driver is to convert the .WMFW firmware file and associated .BIN coefficient (tuning, config) files into a fw_img_v2 image (see Appendix A—fw_img_v2 Specification) header/source file. During boot, the driver can decode the fw_img_v2 contents for delivery over the control port to the CS40L25. This means the driver isn't tied to any particular fw_img at compile time, and instead different, and indeed multiple, fw_imgs can be loaded at runtime.

firmware_converter.py accepts a .WMFW file and zero or more .BIN files for the CS40L25, converting their contents to the C header file cs40l25_fw_img.h. The usage output for 'firmware_converter' is given below.

When needing to incorporate new firmware/tuning/wave table files into the driver, the developer calls firmware_converter.py to generate the new fw_img header/source.

More information on firmware_converter.py usage and building the driver with its output can be found in Convert a WMFW and WMDRs to fw_img.

Usage for firmware_converter

```
$ python3 tools/firmware_converter/firmware_converter.py -h

firmware_converter
Convert from WMFW/WMDR ("BIN") Files to C Header/Source
Version 3.1.0
usage: firmware_converter.py [-h] [--wmdr [WMDRS [WMDRS ...]]] [-s SUFFIX]
                             [-i I2C_ADDRESS] [-b BLOCK_SIZE_LIMIT]
                             [--sym-input SYMBOL_ID_INPUT]
                             [--sym-output SYMBOL_ID_OUTPUT] [--binary]
                             [--wmdr-only] [--generic-sym]
                             [--fw-img-version FW_IMG_VERSION]
                             {print,export,wisce,fw_img_v1,fw_img_v2,json}
                             {cs35l41,cs40l25,cs40l30,cs48l32,cs40l25,cs47l66,cs47l15}
                             wmfw

Parse command line arguments

positional arguments:
  {print,export,wisce,fw_img_v1,fw_img_v2,json}
                        The command you wish to execute.
  {cs35l41,cs40l25,cs40l30,cs48l32,cs40l25,cs47l66,cs47l15}
                        The part number that the wmfw is targeted at.
  wmfw                  The wmfw (or 'firmware') file to be parsed.

optional arguments:
  -h, --help            show this help message and exit
  --wmdr [WMDRS [WMDRS ...]]
                        The wmdr (or 'bin') file(s) to be parsed.
  -s SUFFIX, --suffix SUFFIX
                        Add a suffix to filenames, variables and defines.
  -i I2C_ADDRESS, --i2c-address I2C_ADDRESS
                        Specify I2C address for WISCE script output.
  -b BLOCK_SIZE_LIMIT, --block-size-limit BLOCK_SIZE_LIMIT
                        Specify maximum byte size of block per control port
                        transaction.
  --sym-input SYMBOL_ID_INPUT
                        The location of the symbol table C header(s). If not
                        specified, a header is generated with all controls.
  --sym-output SYMBOL_ID_OUTPUT
                        The location of the output symbol table C header. Only
                        used when no --sym-input is specified.
  --binary              Request binary fw_img output format.
  --wmdr-only           Request to ONLY store WMDR files in fw_img.
  --generic-sym         Use generic algorithm name for 'FIRMWARE_*' algorithm
                        controls
  --fw-img-version FW_IMG_VERSION
                        Release version for the fw_img that ties together a
                        WMFW fw revision with releases of BIN files. Accepts
                        type int of any base.
```

4.2.1 Output as C Source Code

When using the fw_img_v2 command, the default output is the fw_img_v2 block in a C header/source file `cs40l25_fw_img.h` and `cs40l25_fw_img.c` output to the current folder. The source file contains a little-endian byte array of the fw_img_v2 block contents. This source file can be compiled to be linked into the host MCU firmware image. Included in the comment section at the top of the file is the `firmware_converter` command line call used to generate the file along with the version of the tool used. The fw_img_v2 block byte array is also commented to note the sections of the fw_img_v2 specification.

4.2.2 Output as Binary File

The fw_img_v2 block also be exported to a binary format for saving to a ROM external to the host MCU. The binary format contains the little-endian byte array of the fw_img_v2 block as outlined in Appendix A—fw_img_v2 Specification. The `--binary` argument must be added to the command line call. The binary file is then saved to `cs40l25_fw_img.bin`. This binary file can then be used by third-party ROM programmers to save the file to the ROM external to the host MCU.

4.2.3 Changing the Block Size

The fw_img block contains the payload for the `.WMFW` and `.BIN` file contents that must be transferred over the control port to the CS40L25. The contents for both of these files are split among a number of "data blocks" that are arrays of 32-bit words to be written starting at a specific control port address. The number of these data blocks is specified in the fw_img block header member `DATA_BLOCKS`. For the CS40L25, the maximum payload size for these data blocks is 4140 bytes, not including the control port address. For many end systems, there are limits on RAM for buffering a single fw_img data block to transfer out the control port. Therefore, the `firmware_converter` tool offers an option to specify the maximum size of the data blocks when converting to the fw_img block format. This option is specified using the **--block-size-limit** argument.

4.2.4 Separating WMFW and BIN files into separate fw_img_v2

Some end systems may have use cases that require having the `.WMFW` firmware contents and `.BIN` tuning and wavetable contents converted to separate fw_img blocks. Possible examples could include having multiple wavetables that are accessed depending on system use case. In such a case, the firmware and wavetable contents would need to be separated to avoid duplicating the firmware payload. To accomplish this, the `firmware_converter` tool provides the `--wmdr-only` argument that produces a fw_img block containing only the payload contents of the supplied `.BIN` files. To provide a complete set of fw_img blocks, the tool must be called multiple times, as shown in the example below.

```
~/mcu-drivers/cs40l25/fw_8_12_2$ python3 ../../../tools/firmware_converter/firmware_converter.py
fw_img_v2 cs40l25 ../prince_haptics_ctrl_ram_remap_clab_0A0603.wmfw --sym-
input ../../cs40l25_sym.h --generic-sym --wmdr-only --wmdr ../default_clab.bin --suffix clab

firmware_converter
Convert from WMFW/WMDR ("BIN") Files to C Header/Source
Version 3.2.0

Command: fw_img_v2
Part Number: cs40l25
WMFW Path: ../prince_haptics_ctrl_ram_remap_clab_0A0603.wmfw
WMDR Path: ../default_clab.bin
Suffix: clab
Input Symbol ID Header: ../../cs40l25_sym.h
Exported to files:
cs40l25_clab_fw_img.h
cs40l25_clab_fw_img.c

Exit.
```

There are a few important things to note when using the `--wmdr-only` option:

- The main firmware fw_img should be generated as normal.
- Though the firmware contents are not included in the output, the firmware file must still be specified using the `--wmfw` argument to resolve data block addresses.
- It is helpful to use the `--suffix` argument which is inserted into the output filename and exported C file symbols in the C header.

For an example of integrating such a use case into the Board Support Package (BSP) abstraction layer, see the CS35L41 driver Board Support Package (BSP) example code in `cs35l41/bsp/hw_0_bsp_cs35l41.c` function `bsp_dut_boot()`.

5 Example Projects

The MCU Driver Software Package contains example projects for the systems below that integrate the CS40L25 MCU driver.

- Baremetal (no OS) using Internal Boost—see `cs40l25/baremetal/main.c`
- Baremetal (no OS) using External Boost—see `cs40l25/baremetal/main_ext_boost.c`
- FreeRTOS—see `cs40l25/freertos/main.c`

For driver development and testing on a target MCU, the ARM Cortex-M4F-based STM32F401RE is used as the embedded MCU platform. Specifically, the ST NUCLEO-F401RE evaluation system is used connected directly to a CS40L25subsystem. A block diagram of the development and test system can be seen below.

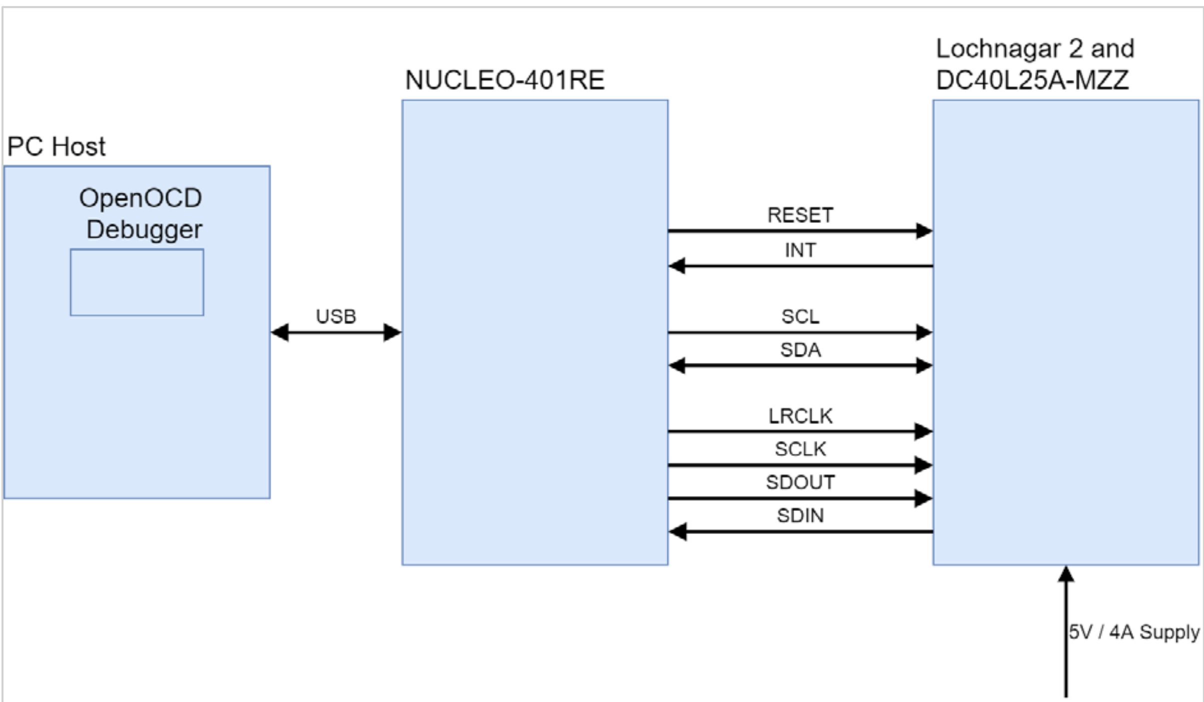

Figure 4 MCU Driver Development Test/Test Hardware Block Diagram

The source files for the running the CS40L25 MCU driver on the development and test hardware are detailed in the table below.

Table 36 BSP Source File Description Table

Folder	Filename	Description
/common/system_test_hw_0/	hw_0_bsp.c hw_0_bsp.h	ST NUCLEO-401RE Board Support Package (BSP) implementation, including implementation of the BSP-to-Driver interface defined in `common/bsp_driver_if.h`
	stm32f4xx_hal_conf.h	Configuration of the STM32F4xx HAL layer
	stm32f4xx_it.cstm32f4xx_it.h	Interrupt handlers
	syscalls.c sysmem.c	Implementation of system calls
	STM32F401RETX_FLASH.ld	GCC Linker script
/cs40l25/bsp/	hw_0_bsp_cs40l25.c hw_0_bsp_dut.h	Implementation of Haptic-specific Board Support Package (BSP) interface required for the 'baremetal' and 'freertos' example projects
/cs40l25/baremetal/	main.c main_ext_boost.c	Implementation of `main()` and application layer callback for the 'baremetal' example project
/cs40l25/freertos/	main.c	Implementation of `main()`, application layer callback, and all RTOS

Folder	Filename	Description
		threads for the 'freertos' example project
	FreeRTOSConfig.h	Application specific definitions for FreeRTOS

Though implemented differently, the 'baremetal' and 'freertos' example projects execute each CS40L25 MCU driver public API abstracted through the Board Support Package (BSP) layer. Calls to the relevant Board Support Package (BSP) layer are spaced out among states in a simple finite state machine, with transitions between states taking place triggered on presses of the 'User' push button on the ST NUCLEO-F401RE. The sequence of states roughly follows the diagram below.

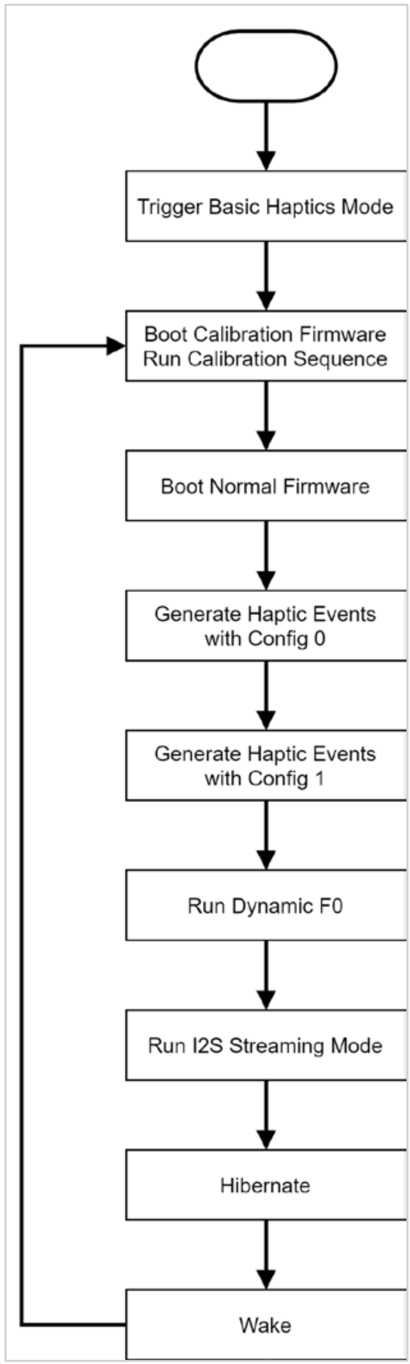

Figure 5 Example CS40L25 Project Flow Diagram

5.1 Development System Requirements

To build the CS40L25 MCU driver, a development system will need the utilities listed below.

- Unzip—any archiving utility able to decompress a .ZIP file (for uncompressing Doxygen documentation archive)
- Bash shell—shell must include 'rm' command line utility
- make—any port of GNU make
- GNU cross-compiler toolchain for the embedded target core.
- Native port of GCC toolchain
- Python 3.x (no external libraries required)

Each example of building the library source will need to be preceded by the following steps:

1. Open a Bash shell
2. Add the location of the cross-compiler to the PATH shell variable
3. Change directory to the unpackaged folder
4. Change directory to `cs40l25` subfolder

5.2 Building the Example Projects

The following examples show the build commands for building the driver library for cortex-m4, as well as the Baremetal and FreeRTOS example projects provided as part of the CS40L25 MCU driver for the ST NUCLEO-F401RE. Much information is given as part of the build output, such as compiler, assembler, and linker flags used, relevant module paths, and all objects to be built.

Following these directions, either a static library `.a` file will be created to link in with a separate MCU host software project, or an example project binary `.ELF` file will be created to run on a test system. This section does not cover loading an `.ELF` onto a target system or debugging MCU code.

5.2.1 Building the Driver Library

Follow the steps below in order to build the driver `.a` static library.

1. `make clean`
2. `make driver_lib_cm4`

5.2.2 Building the Baremetal Example Project

Follow the steps below in order to build the Baremetal example project `.ELF` binary.

1. `make clean`
2. `make baremetal`

5.2.3 Building the FreeRTOS Example Project

Follow the steps below in order to build the FreeRTOS example project `.ELF` binary.

1. `make clean`
2. `make freertos`

6 Appendix A—fw_img_v2 Specification

The fw_img_v2 format is a simple and light-weight way of storing only the contents of a `.wmfw` file and/or coefficients that are needed by a particular project. The format is described below; a fw_img parsing module is provided, along with an example of how to use it.

Header			
Name	**Size**	**Example Value**	**Description**
IMG_MAGIC_NUMBER_1	32-bit	0x54b998ff	First of two magic numbers.
IMG_FORMAT_REV	32-bit	0x2	The format revision of this fw_img, so that parsers can confirm they are compatible.
IMG_SIZE	32-bit	-	The size of the whole image, including both magic numbers and checksum, in bytes.
SYM_TABLE_SIZE	32-bit	10	The number of symbols in the table.
ALG_LIST_SIZE	32-bit	4	The number of algorithm IDs that are present in the firmware.
FW_ID	32-bit	0x1400cb	The ID of the firmware in the image.
FW_VERSION	32-bit	0x90002	The version of the firmware in the image.
DATA_BLOCKS	32-bit	6	The number of data regions to be written out to the device.
MAX_BLOCK_SIZE	32-bit	4140	The maximum size of any single payload block in the fw_img. This can be used to alloc'ing memory during fw_img parsing.
FW_IMG_VERSION	32-bit	0x00010203	An arbitrary version number that can be assigned on fw_img creation.
Symbol Linking Table			
Name	**Size**	**Example Value**	**Description**
SYM_ID_0	32-bit	0x1	The ID for the symbol. A master list of symbol names and IDs will be maintained per-part. The master list should never be edited, only added to, to encourage backwards compatibility. Only symbols in the master list will be extracted from the firmware and put into the symbol table.
SYM_ADDR_0	32-bit	0x2801510	The absolute memory address that the symbol resides at. This will typically be in unpacked memory.
SYM_ID_1	32-bit	0x4	
SYM_ADDR_1	32-bit	0x2801514	
...			
SYM_ID_[SYM_TABLE_SIZE-1]	32-bit	0x43	
SYM_ADDR_[SYM_TABLE_SIZE-1]	32-bit	0x2801518	
Algorithm ID List			
Name	**Size**	**Example Value**	**Description**
ALG_ID_0	32-bit	0xbd	The algorithm ID, as extracted from the firmware.
ALG_ID_1	32-bit	0x117	
...			
ALG_ID_[ALG_LIST_SIZE-1]	32-bit	0x43	
Payload Data			
Name	**Size**	**Example Value**	**Description**
BLOCK_SIZE_0	32-bit	5268	The size of block 0, to be written out to the device, in bytes.
BLOCK_ADDR_0	32-bit	0x2801510	The address at which block 0 should start to be written.
BLOCK_PAYLOAD_0	BLOCK_SIZE_0 bytes	-	The actual data to be written out to the device, formatted to be ready to write out over the bus.
BLOCK_SIZE_1	32-bit	450	

Payload Data			
Name	**Size**	**Example Value**	**Description**
BLOCK_ADDR_1	32-bit	0x2806514	
BLOCK_PAYLOAD_1	BLOCK_SIZE_1 bytes	-	
...			
BLOCK_SIZE_[DATA_BLOCKS-1]	32-bit	360	
BLOCK_ADDR_[DATA_BLOCKS-1]	32-bit	0x2807514	
BLOCK_PAYLOAD_[DATA_BLOCKS-1]	BLOCK_SIZE_[DATA_BLOCKS-1] bytes	-	
Footer			
Name	**Size**	**Example Value**	**Description**
IMG_MAGIC_NUMBER_2	32-bit	0x936be2a6	Second of two magic numbers.
IMG_CHECKSUM	32-bit	0x32f7d9b1	Fletcher-32 of the full image, from IMG_MAGIC_NUMBER_1 to IMG_MAGIC_NUMBER_2.

7 Revision History

Revision	Changes
R1 MAY 2021	• Initial release. Describes Cirrus Logic MCU Driver SDK v4.1.0.

Contacting Cirrus Logic Support
For all product questions and inquiries, contact a Cirrus Logic Sales Representative.
To find the one nearest you, go to www.cirrus.com.