Vol 55, No 3—August 2021

Understanding and Using the No-OS and Platform Drivers

Mahesh Phalke, Senior Software Engineer

Rapidly advancing technology needs software support (firmware drivers and example code) to simplify the design in process. This article describes the use of no-OS (no operating system) drivers and platform drivers for building an application firmware with Analog Devices precision analog-to-digital converters and digital-to-analog converters, which offer a high level of performance in terms of speed, power, size, and resolution.

Embedded firmware examples based on no-OS drivers are provided by ADI to support precision converters. No-OS drivers are responsible for device configuration, data capture from the converter, performing the calibration, etc., while firmware examples based on the no-OS drivers facilitate the transfer of data to a host PC for display, storage, and further processing.

Introduction to No-OS and Platform Drivers

As the name suggests, no-OS drivers are designed for use with generic (or no specific) operating systems. The name also implies that these drivers can be used on BareMetal systems without any OS support. No-OS drivers are designed to provide high-level APIs for digital interface access for the given precision converter. No-OS drivers using these APIs interface with devices to access, configure, read, and write data without knowing about the register addresses (memory map) and their contents.

No-OS drivers make use of the platform driver layer to allow the reuse of the same no-OS drivers across multiple hardware/software platforms, making your firmware highly portable. The use of a platform driver layer insulates the no-OS drivers from knowing about the low-level details of platform specific interfaces such as SPI, I²C, GPIO, etc., which makes the no-OS drivers reusable across multiple platforms without changing them.

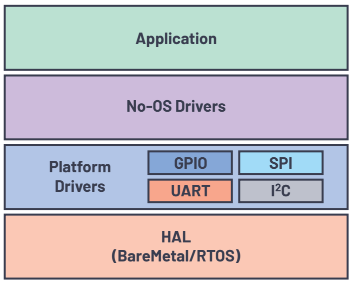
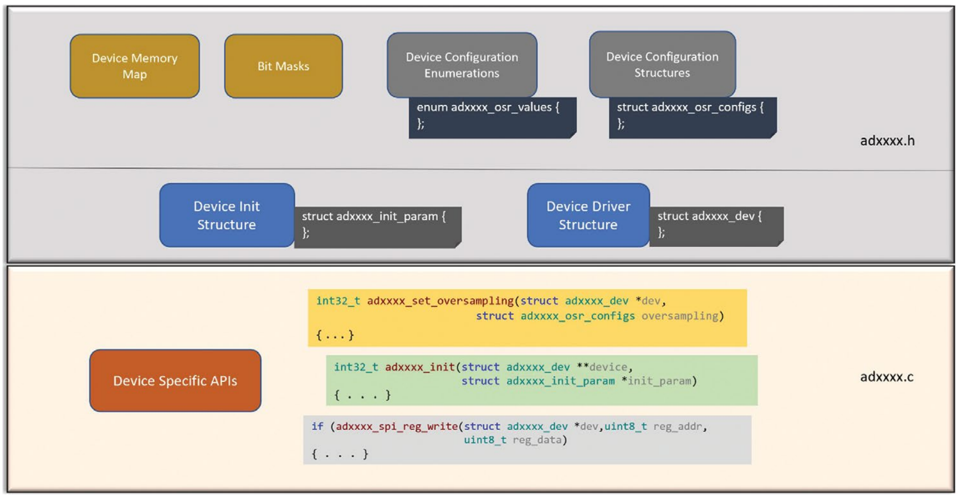


Figure 1. Precision converters firmware stack.

Using No-OS Drivers

Figure 2 shows the typical code structure of no-OS drivers.



```
enum adxxxx_osr_values {
};
```

```
struct adxxxx_osr_configs {
};
```

```
struct adxxxx_init_param {
};
```

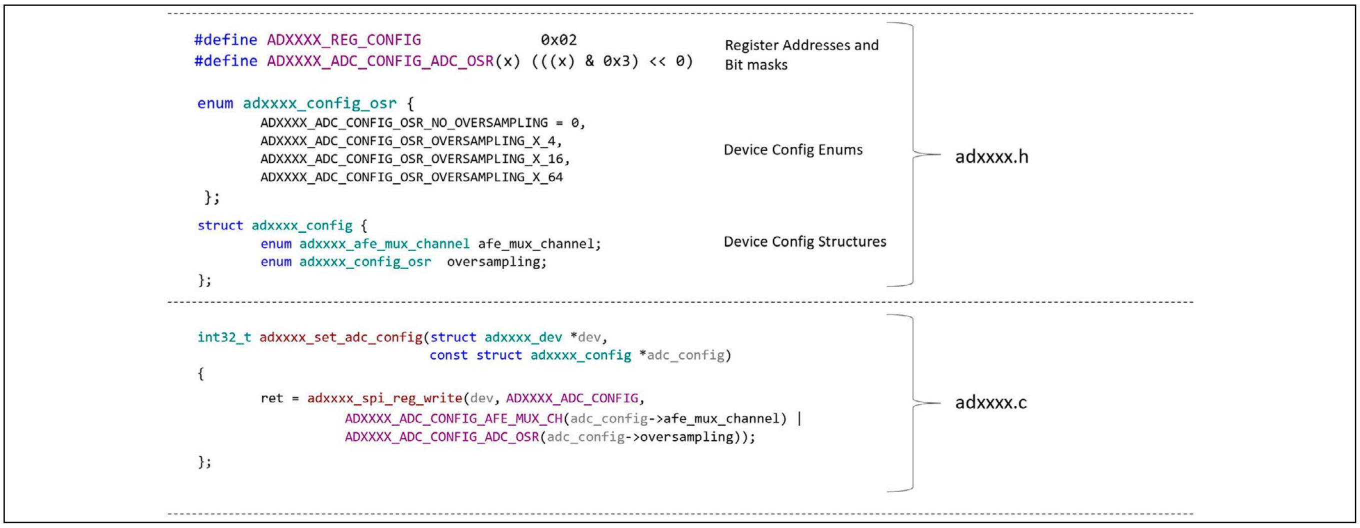
```
struct adxxxx_dev {
};
```

```
int32_t adxxxx_set_oversampling(struct adxxxx_dev *dev,
                                struct adxxxx_osr_configs oversampling)
{...}
```

```
int32_t adxxxx_init(struct adxxxx_dev **device,
                    struct adxxxx_init_param *init_param)
{ . . . }
```

```
if (adxxxx_spi_reg_write(struct adxxxx_dev *dev,uint8_t reg_addr,
                         uint8_t reg_data)
{ . . . }
```

Figure 2. No-OS driver code structure.



```
#define ADXXXX_REG_CONFIG              0x02
#define ADXXXX_ADC_CONFIG_ADC_OSR(x) (((x) & 0x3) << 0)

enum adxxxx_config_osr {
        ADXXXX_ADC_CONFIG_OSR_NO_OVERSAMPLING = 0,
        ADXXXX_ADC_CONFIG_OSR_OVERSAMPLING_X_4,
        ADXXXX_ADC_CONFIG_OSR_OVERSAMPLING_X_16,
        ADXXXX_ADC_CONFIG_OSR_OVERSAMPLING_X_64
};

struct adxxxx_config {
        enum adxxxx_afe_mux_channel afe_mux_channel;
        enum adxxxx_config_osr  oversampling;
};
```

```
int32_t adxxxx_set_adc_config(struct adxxxx_dev *dev,
                              const struct adxxxx_config *adc_config)
{
        ret = adxxxx_spi_reg_write(dev, ADXXXX_ADC_CONFIG,
                ADXXXX_ADC_CONFIG_AFE_MUX_CH(adc_config->afe_mux_channel) |
                ADXXXX_ADC_CONFIG_ADC_OSR(adc_config->oversampling));
};
```

Figure 3. Device configuration enums, structures, and APIs.

The no-OS driver code for precision converters is typically incorporated within two source files written in C programming language: **adxxxx.c** and **adxxxx.h**, where xxxx stands in for the device name (for example, AD7606, AD7124, etc.). The device header file (**adxxxx.h**) contains the public programming interface of device-specific structures, enumerations, register addresses, and bit masks, which are available for public access by including this file into required source files. The device source file (**adxxxx.c**) contains the implementation of the interface used to initialize and remove the device, read/write the device registers, read data from device, get/set device specific parameters, etc.

Typical no-OS drivers are structured around a common set of functionalities:

- The declaration of device specific register addresses, bit mask macros, device configuration enumerations, and structures to read/write the device's specific parameters (for example, oversampling, gain, reference, etc.).
- Initializing/de-initializing the physical device through the no-OS driver's device initialize/remove functions and device specific init and driver structures and descriptors.
- Accessing the device memory map or register details using the device register read/write functions; for example, **adxxxx_read_register()** or **adxxxx_write_register()**.

No-OS Driver Code Usage

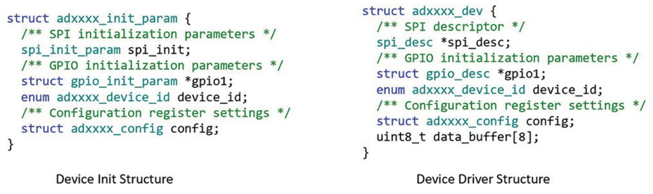
Using device specific addresses, bit masks, and parameter configuration enums and structures:

As stated earlier, the **adxxxx.h** header file contains the declaration of all device specific enums and structures, which are passed to device-specific functions or APIs to configure or access device parameters. This is illustrated in Figure 3.

The **adxxxx_config** structure shown in Figure 3 allows users to select the multiplexer channel and set the oversampling rate for that. Both the members of this structure (**afe_mux_channel** and **oversampling**) are enumerations that are present in the same header file, which contains the numeric constants of all the possible values for both the fields that the user can select.

The **adxxxx_set_adc_config()** function defined in the **adxxxx.c** file takes user-passed configurations/parameters through the configuration structure and further calls the **adxxxx_spi_reg_write()** function to write the data into the **ADXXXX_REG_CONFIG** device register through the digital interface (in the previous case, SPI).

Initializing the device using the no-OS driver device structures and initialization function:



```
struct adxxxx_init_param {
  /** SPI initialization parameters */
  spi_init_param spi_init;
  /** GPIO initialization parameters */
  struct gpio_init_param *gpio1;
  enum adxxxx_device_id device_id;
  /** Configuration register settings */
  struct adxxxx_config config;
}
```

```
struct adxxxx_dev {
  /** SPI descriptor */
  spi_desc *spi_desc;
  /** GPIO initialization parameters */
  struct gpio_desc *gpio1;
  enum adxxxx_device_id device_id;
  /** Configuration register settings */
  struct adxxxx_config config;
  uint8_t data_buffer[8];
}
```

Figure 4. Device init and driver structures' declaration.

Along with the device configuration enumerations and structures, the no-OS drivers provide two additional structures:

- Device initialization structure.
- Device driver structure.

The device init structure allows users to define the device specific parameters and configurations in user application code. The init structure contains the members of other device-specific parameter structures and enumerations. Figure 5 shows how the device init structure is defined.

The device driver structure loads the device initialization parameters through the device init function **adxxxx_init()**. The device driver structure is allocated at run-time (dynamic) memory from the heap space. The parameters declared within the device driver structure and device init structure are almost identical to each other. The device driver structure is a run-time version of the device init structure.

The typical device initialization function and initialization flow is described in the following steps:

- Step 1: Create a definition (or instance) of the device init structure in your application (for example, **struct adxxxx_init_params**) to initialize the user specific device parameters and platform-dependent driver parameters. The parameters are defined during compilation time.

  Note: The parameters defined in the init structure vary from device to device.

  ```
  struct adxxxx_init_param adxxxx_init_params = { … }
  ```

- Step 2: Create a pointer instance (variable) of the device driver structure in the application code.

  A user application needs to create a single pointer instance of the device driver structure. This instance is passed to all no-OS driver APIs/functions to access device specific parameters. This pointer instance defined in the application code points to a dynamically allocated memory in heap, which is done through a device init function such as **adxxxx_init()**, which is defined in the no-OS drivers.

  ```
  static struct adxx_dev *p_adxxxx_dev = NULL;
  ```

- Step 3: Initialize the device and other platform specific peripherals by calling the device init function.

  ```
  /* Initialize ADxxxx device and peripheral interface */
  init_status = adxxxx_init(&p_adxxxx_dev, &adxxxx_init_str);
  if (init_status != SUCCESS) {
          return init_status;
  }
  ```

  The **adxxxx_init()** function defined in the no-OS drivers initializes the device with the user specific parameters passed through **adxxx_init_param** structure. The pointer instance of the device driver structure and the instance of the device init structure are passed as two arguments to this init function. The user application code can call the **adxxxx_init()** function multiple times, provided the init call is balanced by a call to the device remove function.

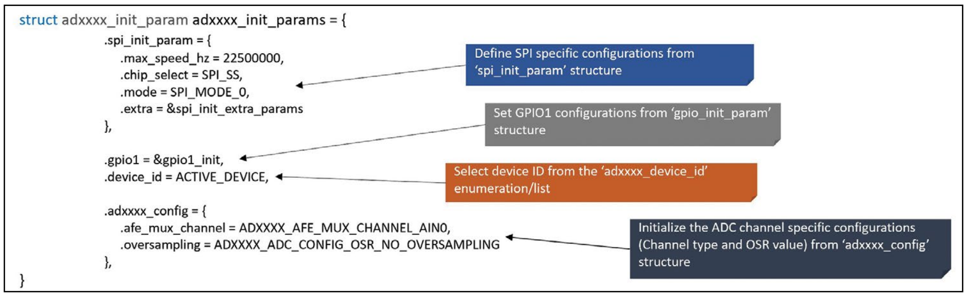
Accessing the memory map (register contents) through device register read/write functions is shown in Figure 6

Users can access the device register contents (such as the product ID, scratch pad value, OSR, etc.) through no-OS driver device-specific **adxxx_read/write()** functions.

Most of the time, users don't use register access functions directly. The device specific functions call through these register access functions such as **adxxxx_spi_reg_read/write()**. The use of device configuration and status APIs to access the device memory map is recommended when possible instead of using the direct register access functions, as this ensures the device driver structures are kept in sync with the configuration in the device.

Platform Drivers

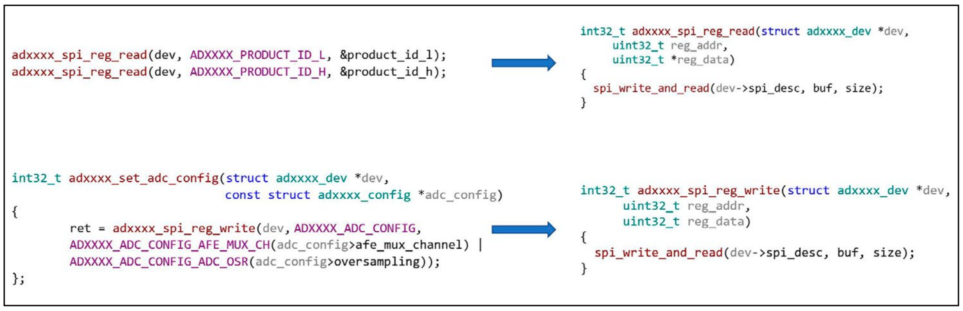
Platform drivers are one of the hardware abstraction layers (HAL) that wrap the platform specific APIs. They are called by no-OS device drivers or user application code to provide an independence from the underlying hardware and software platforms. Platform drivers wrap the low-level platform-specific hardware functionality such as SPI/I²C initialization and read/write, GPIO initialization and read/write, UART initialization and receive/transmit, user specific delays, interrupts, etc.



```
struct adxxxx_init_param adxxxx_init_params = {
        .spi_init_param = {
            .max_speed_hz = 22500000,
            .chip_select = SPI_SS,
            .mode = SPI_MODE_0,
            .extra = &spi_init_extra_params
        },

        .gpio1 = &gpio1_init,
        .device_id = ACTIVE_DEVICE,

        .adxxxx_config = {
            .afe_mux_channel = ADXXXX_AFE_MUX_CHANNEL_AIN0,
            .oversampling = ADXXXX_ADC_CONFIG_OSR_NO_OVERSAMPLING
        },
}
```

Define SPI specific configurations from 'spi_init_param' structure
Set GPIO1 configurations from 'gpio_init_param' structure
Select device ID from the 'adxxxx_device_id' enumeration/list
Initialize the ADC channel specific configurations (Channel type and OSR value) from 'adxxxx_config' structure

Figure 5. The device init structure definition in the user application.



```
adxxxx_spi_reg_read(dev, ADXXXX_PRODUCT_ID_L, &product_id_l);
adxxxx_spi_reg_read(dev, ADXXXX_PRODUCT_ID_H, &product_id_h);
```

```
int32_t adxxxx_spi_reg_read(struct adxxxx_dev *dev,
     uint32_t reg_addr,
     uint32_t *reg_data)
{
  spi_write_and_read(dev->spi_desc, buf, size);
}
```

```
int32_t adxxxx_set_adc_config(struct adxxxx_dev *dev,
                              const struct adxxxx_config *adc_config)
{
        ret = adxxxx_spi_reg_write(dev, ADXXXX_ADC_CONFIG,
        ADXXXX_ADC_CONFIG_AFE_MUX_CH(adc_config>afe_mux_channel) |
        ADXXXX_ADC_CONFIG_ADC_OSR(adc_config>oversampling));
};
```

```
int32_t adxxxx_spi_reg_write(struct adxxxx_dev *dev,
      uint32_t reg_addr,
      uint32_t reg_data)
{
  spi_write_and_read(dev->spi_desc, buf, size);
}
```

Figure 6. Accessing the register contents.

The typical file structure of an SPI platform driver module is shown in Figure 7.

Using Platform Drivers

Platform driver code is typically incorporated within three source files written in C/C++ programming languages.

1) **spi.h:** This is a platform agnostic file that contains the device structures and enumerations required for the SPI functionality. The C programming interface defined in this header has no platform dependencies.

   All the parameters declared within the init and device structures are common to SPI interfaces on any platform.

   The **void *extra** parameter used in device init structure allows users to pass additional (extra) parameters, which could be specific to the platform that is used.

   The parameters declared within the SPI driver structure and SPI init structure are almost identical to each other. The SPI driver structure is a run-time version of the SPI init structure.

2) **spi.cpp/.c:** This file contains the implementation of the functions declared in the **spi.h** file, which are used to initialize the SPI peripheral and read/write data from it for a specific platform. The term "platform," in a broader sense, refers to a combination of a hardware microcontroller (target device) and software (for example, RTOS or Mbed-OS). This file is platform dependent and needs to be modified while porting on some other platform.

Figure 9 details the SPI interfaces for the Mbed platform and shows how to initialize SPI and read/write data using these interfaces and device init/driver structures.

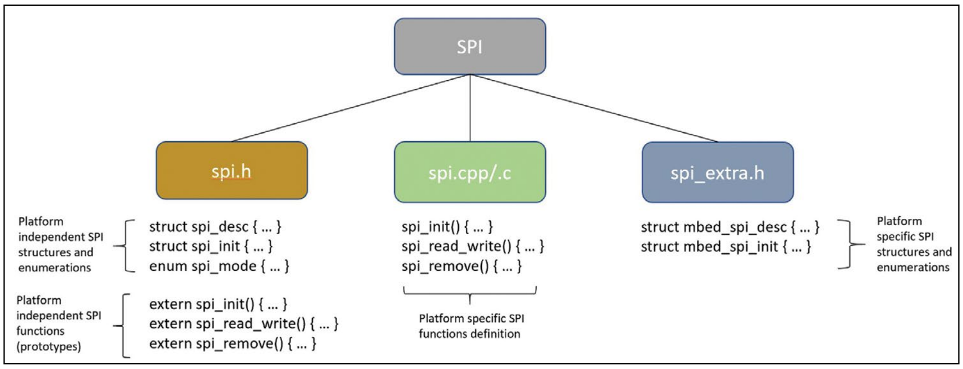
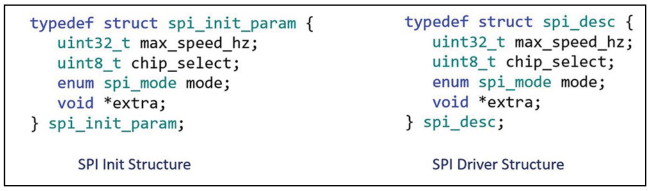

SPI
spi.h
spi.cpp/.c
spi_extra.h
Platform independent SPI structures and enumerations
struct spi_desc { … }
struct spi_init { … }
enum spi_mode { … }
Platform independent SPI functions (prototypes)
extern spi_init() { … }
extern spi_read_write() { … }
extern spi_remove() { … }
spi_init() { … }
spi_read_write() { … }
spi_remove() { … }
Platform specific SPI functions definition
struct mbed_spi_desc { … }
struct mbed_spi_init { … }
Platform specific SPI structures and enumerations

Figure 7. SPI platform driver code structure.



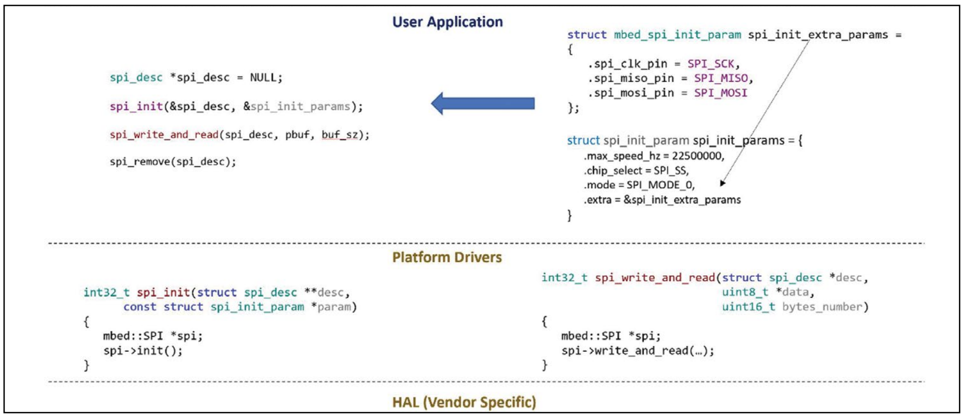
```
typedef struct spi_init_param {
    uint32_t max_speed_hz;
    uint8_t chip_select;
    enum spi_mode mode;
    void *extra;
} spi_init_param;
```

SPI Init Structure

```
typedef struct spi_desc {
    uint32_t max_speed_hz;
    uint8_t chip_select;
    enum spi_mode mode;
    void *extra;
} spi_desc;
```

SPI Driver Structure

Figure 8. SPI init and driver structures.



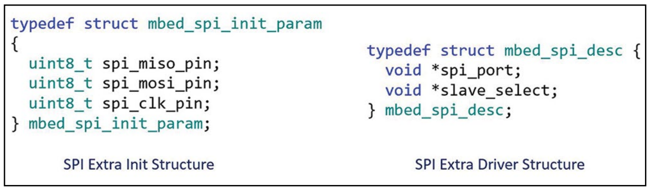
```
spi_desc *spi_desc = NULL;

spi_init(&spi_desc, &spi_init_params);

spi_write_and_read(spi_desc, pbuf, buf_sz);

spi_remove(spi_desc);
```

```
struct mbed_spi_init_param spi_init_extra_params =
{
    .spi_clk_pin = SPI_SCK,
    .spi_miso_pin = SPI_MISO,
    .spi_mosi_pin = SPI_MOSI
};

struct spi_init_param spi_init_params = {
   .max_speed_hz = 22500000,
   .chip_select = SPI_SS,
   .mode = SPI_MODE_0,
   .extra = &spi_init_extra_params
}
```

```
int32_t spi_init(struct spi_desc **desc,
     const struct spi_init_param *param)
{
   mbed::SPI *spi;
   spi->init();
}
```

```
int32_t spi_write_and_read(struct spi_desc *desc,
                           uint8_t *data,
                           uint16_t bytes_number)
{
   mbed::SPI *spi;
   spi->write_and_read(…);
}
```

Figure 9. SPI APIs or functions. Note: the added code for spi_init() and spi_write_and_read() is abbreviated code and details are omitted for clarity.



```
typedef struct mbed_spi_init_param
{
  uint8_t spi_miso_pin;
  uint8_t spi_mosi_pin;
  uint8_t spi_clk_pin;
} mbed_spi_init_param;
```

SPI Extra Init Structure

```
typedef struct mbed_spi_desc {
  void *spi_port;
  void *slave_select;
} mbed_spi_desc;
```

SPI Extra Driver Structure

Figure 10. SPI extra init and driver structures.

3) **spi_extra.h:** This file contains the additional device structures or enumerations, which are specific to a given platform. This allows user application code to provide the configurations that are not covered in a generic **spi.h** file. For example, the SPI pins can vary from platform to platform, which therefore can be added as part of these platform specific extra structures.

Porting Platform Drivers

The platform drivers can be ported from one platform (microcontroller) to another by typically creating platform specific **.cpp/.c** and **_extra.h** files. Platform drivers reside one layer above the device-specific hardware abstraction layer (HAL) provided by the microcontroller unit vendor. Therefore, porting of platform drivers from one platform to another needs some minimal changes in platform driver code related to calling the functions or APIs present in the HAL provided by its vendor.

The diagram in Figure 12 distinguishes between Mbed-based SPI platform drivers and ADuCM410 SPI platform drivers.

The GitHub source code links for ADI's no-OS repository and platform drivers are available on the Analog Devices Wiki and GitHub page.

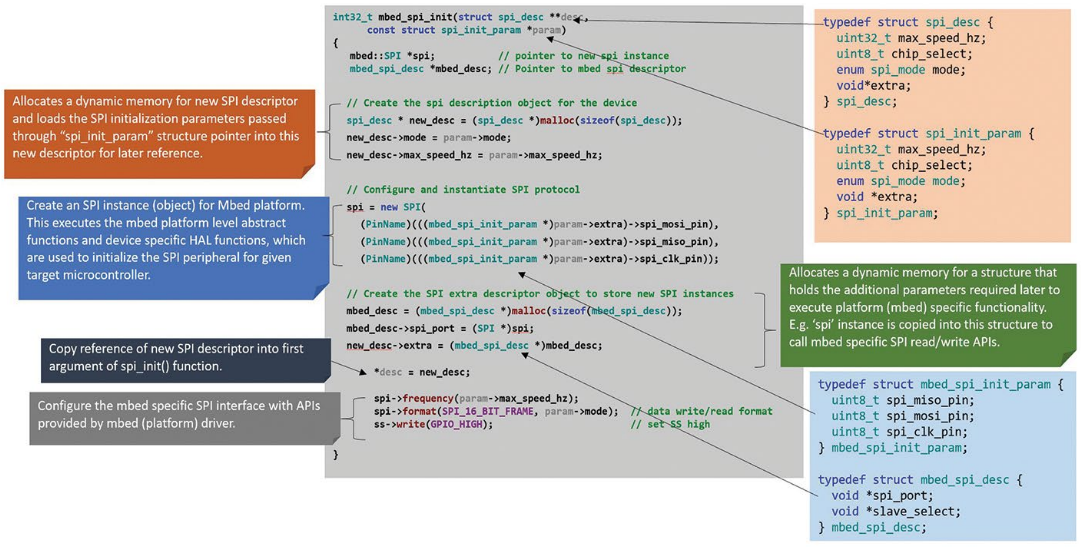
Contributing to No-OS Drivers

ADI no-OS drivers are open source and hosted on GitHub. The drivers not only support precision converters, but also many other Analog Devices products such as accelerometers, transceivers, photoelectronic devices, and more. Anyone familiar with the source code can contribute to these drivers by committing the changes and creating a pull request to review those changes.

There are many example projects that can run in Linux and/or Windows environments. Many example projects are developed with a hardware descriptive language (HDL) to run on FPGAs developed by Xilinx®, Intel®, etc., and that target processors developed by different vendors.

No-OS software drivers in C for systems without an operating system can be accessed at the Analog Devices no-OS GitHub repository.

The Analog Devices Wiki provides examples developed for precision converters using the Mbed and ADuCMxxx platform.

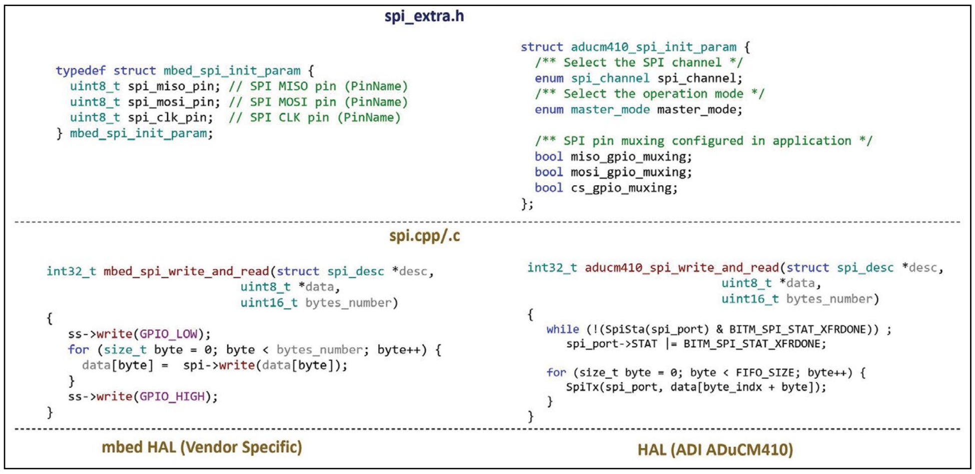

```c
int32_t mbed_spi_init(struct spi_desc **desc,
     const struct spi_init_param *param)
{
   mbed::SPI *spi;            // pointer to new spi instance
   mbed_spi_desc *mbed_desc; // Pointer to mbed spi descriptor

  // Create the spi description object for the device
  spi_desc * new_desc = (spi_desc *)malloc(sizeof(spi_desc));
  new_desc->mode = param->mode;
  new_desc->max_speed_hz = param->max_speed_hz;

  // Configure and instantiate SPI protocol
  spi = new SPI(
     (PinName)(((mbed_spi_init_param *)param->extra)->spi_mosi_pin),
     (PinName)(((mbed_spi_init_param *)param->extra)->spi_miso_pin),
     (PinName)(((mbed_spi_init_param *)param->extra)->spi_clk_pin));

  // Create the SPI extra descriptor object to store new SPI instances
  mbed_desc = (mbed_spi_desc *)malloc(sizeof(mbed_spi_desc));
  mbed_desc->spi_port = (SPI *)spi;
  new_desc->extra = (mbed_spi_desc *)mbed_desc;

     *desc = new_desc;

     spi->frequency(param->max_speed_hz);
     spi->format(SPI_16_BIT_FRAME, param->mode);  // data write/read format
     ss->write(GPIO_HIGH);                        // set SS high

}
```

```c
typedef struct spi_desc {
  uint32_t max_speed_hz;
  uint8_t chip_select;
  enum spi_mode mode;
  void*extra;
} spi_desc;

typedef struct spi_init_param {
  uint32_t max_speed_hz;
  uint8_t chip_select;
  enum spi_mode mode;
  void *extra;
} spi_init_param;
```

```c
typedef struct mbed_spi_init_param {
  uint8_t spi_miso_pin;
  uint8_t spi_mosi_pin;
  uint8_t spi_clk_pin;
} mbed_spi_init_param;

typedef struct mbed_spi_desc {
  void *spi_port;
  void *slave_select;
} mbed_spi_desc;
```

Allocates a dynamic memory for new SPI descriptor and loads the SPI initialization parameters passed through "spi_init_param" structure pointer into this new descriptor for later reference.
Create an SPI instance (object) for Mbed platform. This executes the mbed platform level abstract functions and device specific HAL functions, which are used to initialize the SPI peripheral for given target microcontroller.
Copy reference of new SPI descriptor into first argument of spi_init() function.
Configure the mbed specific SPI interface with APIs provided by mbed (platform) driver.
Allocates a dynamic memory for a structure that holds the additional parameters required later to execute platform (mbed) specific functionality. E.g. 'spi' instance is copied into this structure to call mbed specific SPI read/write APIs.

Figure 11. Mbed platform specific SPI init implementation.



```c
typedef struct mbed_spi_init_param {
  uint8_t spi_miso_pin; // SPI MISO pin (PinName)
  uint8_t spi_mosi_pin; // SPI MOSI pin (PinName)
  uint8_t spi_clk_pin;  // SPI CLK pin (PinName)
} mbed_spi_init_param;
```

```c
struct aducm410_spi_init_param {
  /** Select the SPI channel */
  enum spi_channel spi_channel;
  /** Select the operation mode */
  enum master_mode master_mode;

  /** SPI pin muxing configured in application */
  bool miso_gpio_muxing;
  bool mosi_gpio_muxing;
  bool cs_gpio_muxing;
};
```

```c
int32_t mbed_spi_write_and_read(struct spi_desc *desc,
                                uint8_t *data,
                                uint16_t bytes_number)
{
   ss->write(GPIO_LOW);
   for (size_t byte = 0; byte < bytes_number; byte++) {
     data[byte] =  spi->write(data[byte]);
   }
   ss->write(GPIO_HIGH);
}
```

```c
int32_t aducm410_spi_write_and_read(struct spi_desc *desc,
                                    uint8_t *data,
                                    uint16_t bytes_number)
{
   while (!(SpiSta(spi_port) & BITM_SPI_STAT_XFRDONE)) ;
      spi_port->STAT |= BITM_SPI_STAT_XFRDONE;

   for (size_t byte = 0; byte < FIFO_SIZE; byte++) {
      SpiTx(spi_port, data[byte_indx + byte]);
   }
}
```

Figure 12. Platform driver differences.

About the Author

Mahesh Phalke is a senior software engineer at Analog Devices Bangalore in the Precision Converters Technology Software Group. He graduated from Pune University in 2011 with a bachelor's degree in electronics engineering. He can be reached at mahesh.phalke@analog.com.