Device Drivers User Guide

Device Drivers User Guide
Version 1.0.0, March 2017

http://www.analog.com

Contents

1 User Guide

This document provides the guidelines for using the Analog Devices device drivers included in the Board Support Package. This document is specific to the bus drivers such as SPI, I2C, SPORT and UART. The other drivers such as GPIO, Timers, Accelerometers does not follow a common API model.

1.1 Features Supported by Device Drivers

The device drivers:

- are simple to use;
- have a minimal code and data footprint;
- add minimal run-time overhead;
- do not require memory copying between driver and application;
- do not require dynamic memory allocation by Device drivers;
- support switching between DMA mode and interrupt mode at run-time;
- support reentrancy but are not thread safe (same instance of the driver cannot be used from two different threads)
- are MISRA-C 2012 compliant.

1.2 Device Driver Operating Modes

The drivers operate in one of three operating modes of interaction, which determine how the driver and application interact:

1. Blocking mode. This mode is entered when a blocking API is called.
2. Non-Blocking mode. This mode is entered when a non-blocking API is called where no callback is registered.
3. Callback mode. This mode is entered when a non-blocking API is called where a callback is registered.

These modes are mutually exclusive with one another. An application is not allowed to mix these modes. The modes are described in more detail in the following sections.

Depending on the peripheral, a driver may also have a choice of modes for internal operation:

- Interrupt mode.

- DMA mode.

These modes of internal operation are selected separately from the interaction modes of Blocking, Non-Blocking and Callback mode.

1.2.1 Blocking mode

In blocking mode, a read or write call does not return until the read or write transaction has completed.

When operating in an RTOS environment, a task will yield the processor when making a blocking call. The RTOS will schedule in another task that is ready to run. The blocked task will be placed back on the ready to run queue upon completion of the read or write transaction.

In a non-RTOS environment, there is only one thread of execution. A call to a blocking API will result in the thread "spinning" (or simply, waiting) until the read or write transaction completes.

See *Blocking Mode APIs* for more information on entering blocking mode

1.2.2 Non-Blocking mode for all devices

In non-blocking mode, a read or write call does returns immediately, even if the read or write transaction has not completed yet. The driver will finish the read or write transaction. It is the application's responsibility to synchronize with the transaction completion.

The synchronization mechanism that the application must use is detailed in *Non-blocking Transactions and Buffer Ownership*. A driver is placed into non-blocking mode simply by calling the non-blocking read or write APIs when no callback is registered. See section *Non-blocking Transactions and Buffer Ownership* for information about non-blocking read and write APIs.

1.2.3 Callback mode

Similar to non-blocking mode, in callback mode, a read or write call returns immediately. It does not wait for the transaction to complete. Unlike non-blocking mode, the synchronization mechanism is an application-specified Callback. An application will pass a Callback to the device driver to be called upon completion of a read or write transaction. The device driver executes the callback to an application-specified routine when the read or write transaction completes or when an error occurs.

A driver is placed into callback mode simply by registering a callback with the device driver. More information on Callback mode can be found in *Using Callback Mode*.

1.3 Device Driver API Reference

The following section provides an overview of the device driver APIs. Each API will indicate to which of the three operation modes the API is applicable. Some APIs can be used in all modes, while others are specific to just one or two modes of operation.

Syntactic conventions used in this overview:

Each driver will have API names unique to the controller. For example,

`adi_i2c_Open`

is specific to the I2C controller while:

`adi_spi_Open`

is specific to the SPI controller. However, in this API overview, controller-independent syntax is used to indicate that the API is applicable to all controllers. Therefore,

`adi_xxx_Open`

is used where the "xxx" implies that the API is applicable to all controller drivers.

A number of APIs that have receive and transmit versions of the APIs, indicated by "Rx" and "Tx" in the API names, for example `adi_xxx_GetRxBuffer` and `adi_xxx_GetTxBuffer`. When discussing common behavior of these APIs, the "Rx"/"Tx" portion is omitted, for example `adi_xxx_GetBuffer`.

The following table shows which APIs are valid in the particular device driver operating mode.

	Blocking	Non-Blocking	Callback
adi_xxx_Read/Write	Yes	No	No
adi_xxx_SubmitBuffer	No	Yes	Yes
adi_xxx_GetBuffer	No	Yes	No
adi_xxx_IsBufferAvailable	No	Yes	No
adi_xxx_Close	No	Yes	Yes

In the sequence diagrams, blue represents code executing at the interrupt level and orange represents code executing at the thread level.

1.3.1 Open/Close

adi_xxx_Open

The open function opens the device and returns a *handle* to the device instance. The handle is an abstract/opaque data structure that is unique to the instance of the controller that is being opened. The instance of the controller is indicated by the `nDeviceNum` parameter as shown below. The handle is then passed into all subsequent calls which allows the driver to know on which controller instance the call is operating.

Each device driver requires memory to record information about the state of the driver. This memory must be passed in from the application.The driver indicates the size requirements in the API header file as shown below.

```
/* Memory required for the driver in terms of bytes */
 #define ADI_XXX_MEMORY_SIZE 100

 ADI_XXX_RESULT adi_xxx_Open (
         uint32_t              nDeviceNum,
         ...
         void *const           pDeviceMemory,
         uint32_t              nMemorySize
         ADI_XXX_HANDLE const *phDevice
 );
```

This API is valid for all modes.

adi_xxx_Close

This API closes the given device instance.

```
ADI_XXX_RESULT adi_xxx_Close (
        ADI_XXX_HANDLE const hDevice
);
```

This API is valid for all modes.

1.3.2 Non-Blocking Mode APIs and Buffer Ownership

All memory buffers for read and write transactions must be allocated by the application. The drivers do not perform any dynamic memory allocation. When a read or write transaction is initiated by an application, a memory buffer is passed to the driver.

In the case of a non-blocking transaction, the call to read or write returns immediately. The driver completes the transaction asynchronously. While the driver is completing the transaction, the driver owns the buffer.

The only mechanism to transfer buffer ownership from the driver back to the application is the `adi_xxx_GetBuffer` API (there are Rx and Tx versions of the API). When an application calls this API, the application blocks until the transaction is completed by the driver. If the transaction has already completed, the application returns from this call immediately, regaining ownership of the buffer.

```
ADI_XXX_RESULT adi_xxx_GetBuffer (

        ADI_XXX_HANDLE const hDevice,
        void **        const ppBuffer,
        uint32_t *     const pHwError

);
```

These APIs are valid for Non-Blocking mode only.

1.3.3 Non blocking buffer transaction APIs

The device drivers are designed to have a minimal footprint and minimal latency. The device drivers do not maintain read or write buffer queues. They support only a ping pong buffer or single buffer scheme depending upon the peripheral. The slow speed devices like i2c and SPI support only one outstanding transaction at a time, peripherals like UART and SPORT support ping pong buffer mode (up to two outstanding transaction at a time).

To use the device drivers in non-blocking mode and to stream data to or from the driver, an application must allocate two buffers and then proceed to use them in the following ping pong manner:

```
// Pseudo-code
adi_xxx_SubmitBuffer // transfer buffer to driver
while (cond)
{
    adi_xxx_SubmitBuffer // transfer buffer to driver
    adi_xxx_GetBuffer     // will block until transaction complete
    // buffer is owned by application again
    // buffer processing
}
```

adi_xxx_SubmitBuffer

This API initiates a read transaction by submitting a buffer for reading. The API transfers ownership of the buffer to the driver. The driver retains ownership of the buffer until the application calls `adi_xxx_GetRxBuffer`.

The drivers support up to two outstanding transactions at a time. If more than two transactions are requested, this API returns an error indicating that too many transactions have been requested.

```
ADI_XXX_RESULT adi_xxx_SubmitBuffer (
        ADI_XXX_HANDLE const hDevice,
        void *         const pBuffer,
        uint32_t       const nBufSize,
        bool           const bDMA

 );
```

This API is valid for Non-Blocking and Callback modes.

adi_xxx_GetBuffer

This API permits an application to transfer buffer ownership from the driver back to the application. The buffer is transferred back to the application only after the transaction that the buffer is associated with has completed. If the transaction is not completed yet, the application blocks until the transaction is completed. If the transaction has already completed, the API returns immediately.

In an RTOS environment, waiting for a transaction implies yielding the processor to the next task that is ready to run. In a non-RTOS environment, waiting for a transaction implies "spinning" or simply polling for completion. In a non-RTOS environment "spinning" will prevent any other useful work from occurring.

In a non-RTOS environment, to avoid waiting for completion, applications can use the non blocking peek function `adi_xxx_IsRxBufferAvailable`, and (if the buffer is not available) they can perform other tasks. (This API is also functional in an RTOS environment).

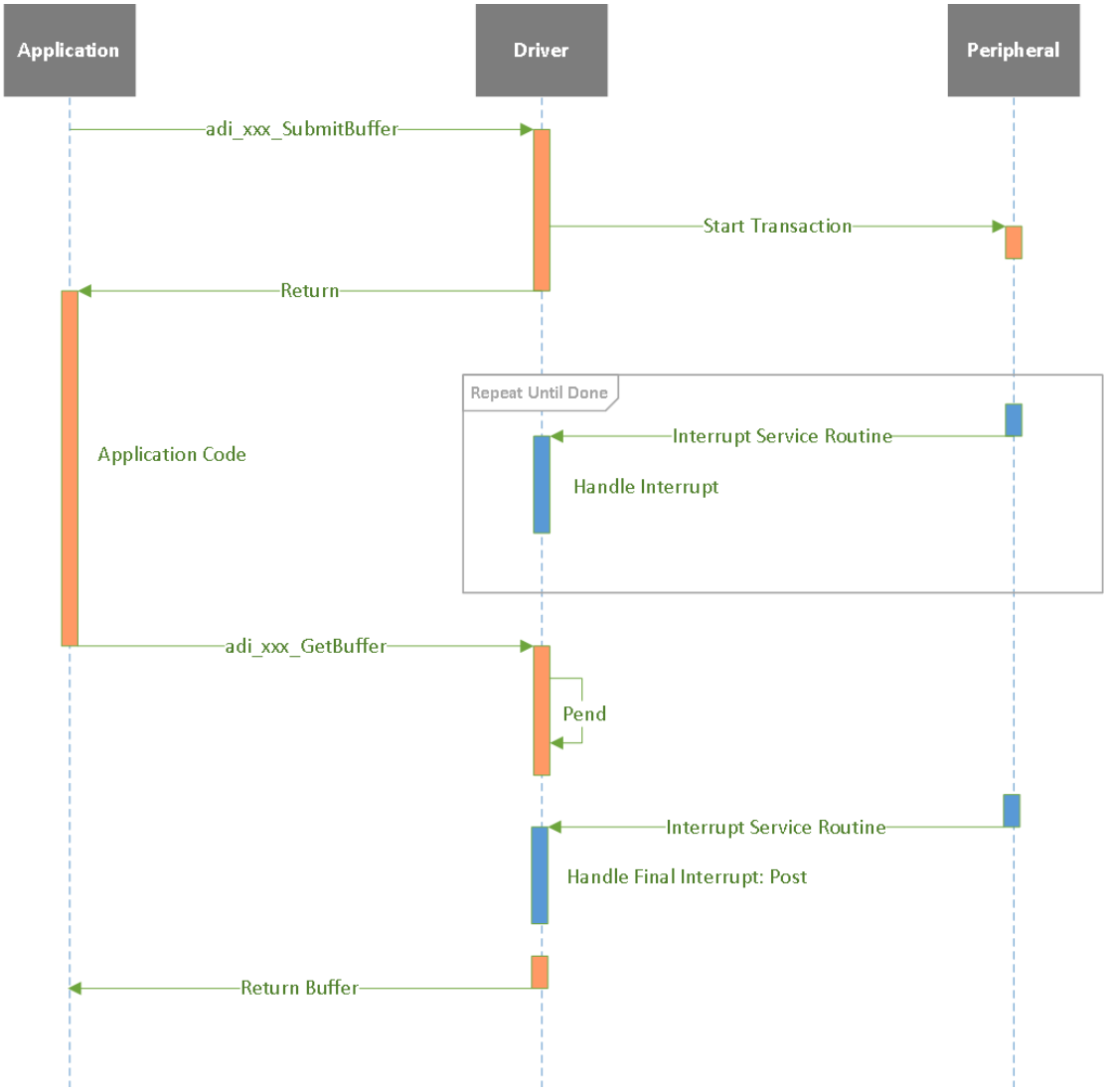
When a peripheral error is detected, this API returns immediately with an error indicating that a hardware error has occurred. The actual hardware error(s) will be written into the user-provided variable pointed to by pHwError, see *Peripheral Error Reporting*, which explains how errors are reported.

```
ADI_XXX_RESULT adi_xxx_GetRxBuffer (
        ADI_XXX_HANDLE const hDevice,
        void **        const ppBuffer,
        uint32_t *     const pHwError
```

```
);
```

This API is valid only for Non-Blocking mode.

The following sequence diagram shows how the non-blocking mode APIs are used to interact with the device driver.



1.3.4 Non-Blocking Peek Functions

These APIs can be used to check if a free buffer is available without blocking. These functions can be used (the whole CPU in the non-RTOS case or the task in the RTOS case) to avoid blocking when the buffer is not available.

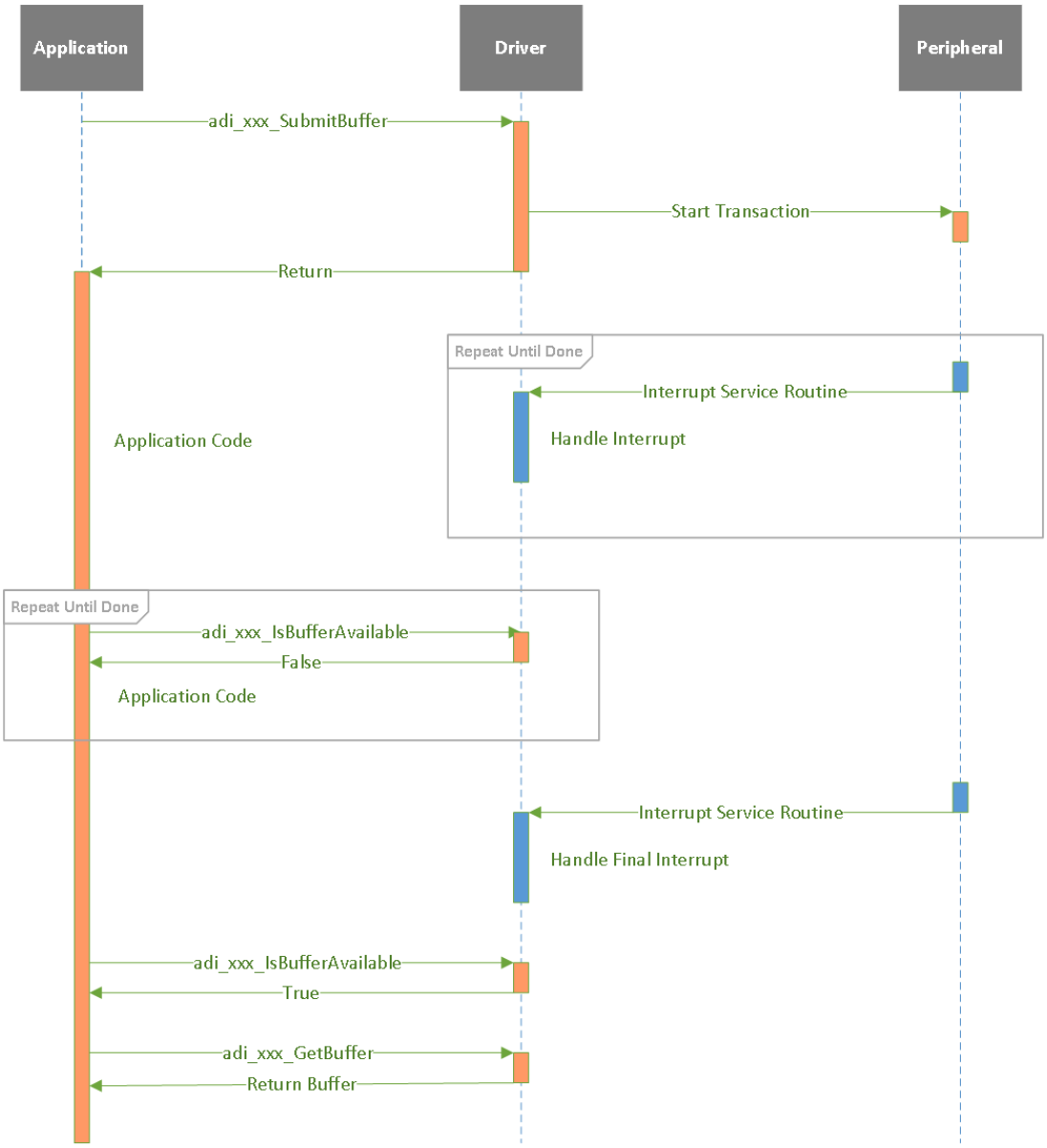
adi_xxx_IsBufferAvailable

Checks if the filled Rx buffer is available for processing.

```
ADI_XXX_RESULT adi_xxx_IsRxBufferAvailable(
        ADI_XXX_HANDLE const hDevice,
        bool * const         pbAvailable
 );
```

This API is valid only for Non-Blocking mode.

The following sequence diagram shows how the non-blocking mode APIs are used to interact with the device driver when using peek functions.



1.3.5 Blocking Mode APIs

By calling these APIs, a driver will be placed into blocking mode. These APIs wait until the given buffer is processed. These APIs are available only for low-speed devices, such as the UART, I2C, and SPI controllers.

adi_xxx_Write

This API submits the given buffer for transmission and waits until it is transmitted.

When a peripheral error is detected, this API returns immediately with an error indicating that a hardware error has occurred. The actual hardware error(s) will be written into the user-provided variable pointed to by pHwError, see *Peripheral Error Reporting*, which explains how errors are reported.

```
ADI_XXX_RESULT adi_xxx_Write (

        ADI_XXX_HANDLE const hDevice,
        void *         const pBuffer,
        uint32_t       const nBufSize,
        bool           const bDMA,
        uint32_t *     const pHwError
);

This API is valid only for Blocking mode.
```

adi_xxx_Read

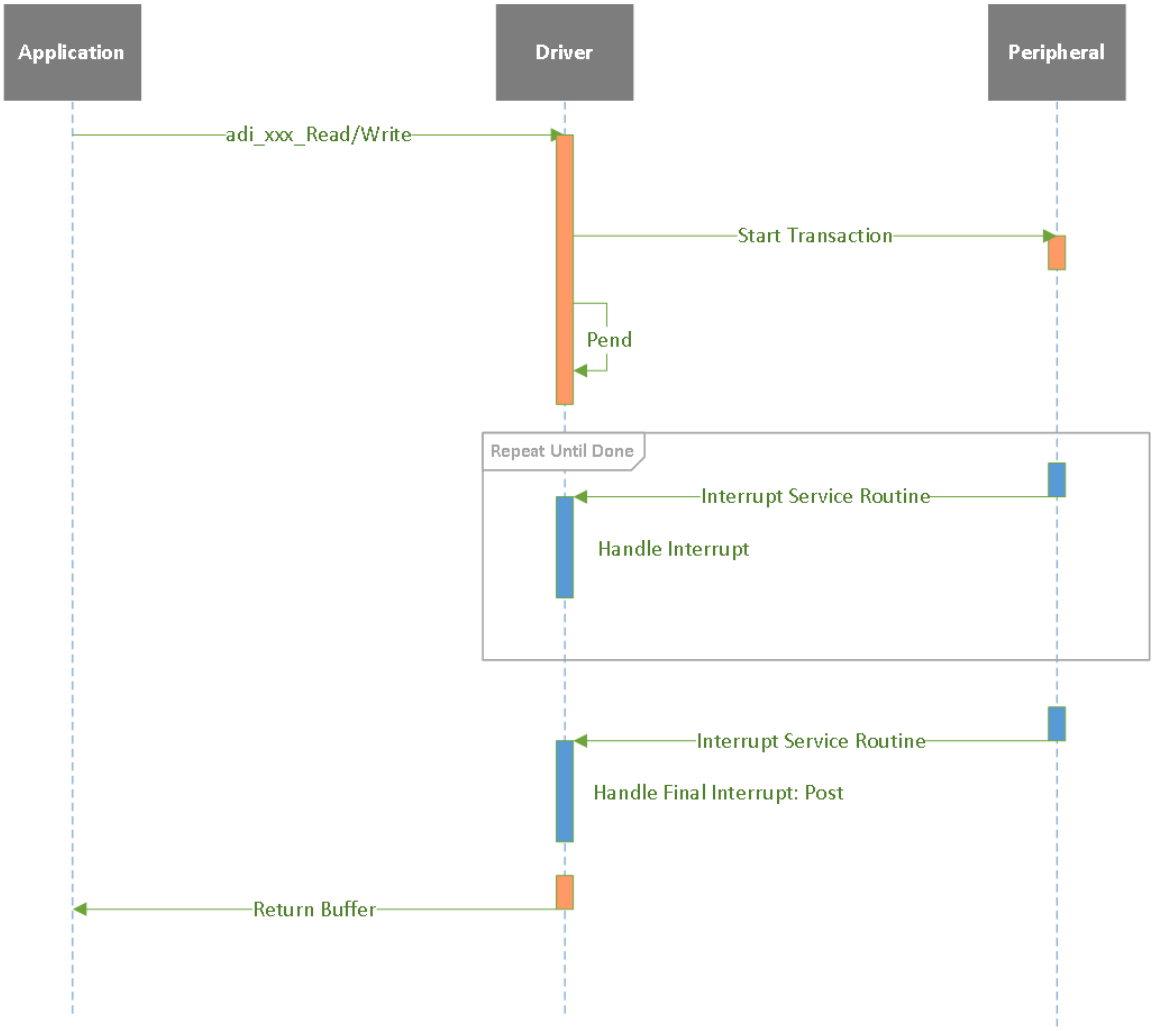
This API submits the given buffer for receiving and waits until the buffer is filled before returning.

When a peripheral error is detected, this API returns immediately with an error indicating that a hardware error has occurred. The actual hardware error(s) will be written into the user-provided variable pointed to by pHwError, see *Peripheral Error Reporting*, which explains how errors are reported.

```
ADI_XXX_RESULT adi_xxx_Read (
        ADI_XXX_HANDLE const hDevice,
        void *         const pBuffer,
        uint32_t       const nBufSize,
        bool           const bDMA,
        uint32_t *     const pHwError
);
```

This API is valid only for Blocking mode.

The following sequence diagram shows how the blocking mode APIs are used to interact with the device driver.



1.3.6 Switching Between Interrupt and DMA Mode

Drivers permit an application to switch between interrupt mode and DMA mode at run time. Interrupt mode can be advantageous to use for short transfers (1 to 2 words), reducing the overhead for setting up a DMA transaction. This API can be useful in scenarios where application is looking for a pattern/header before starting the actual DMA: initially, the application would start the peripheral in interrupt mode and schedules short transfer for interpreting the pattern. After the expected pattern is received, the application switches over to the DMA mode.

Switching between Interrupt and DMA mode can be done on a per transaction basis. The *adi_xxx_SubmitBuffer*, *adi_xxx_Read/Write* APIs take a boolean parameter *bDMA* to allow application to choose if the transaction should be completed using DMA mode or Interrupt mode. When bDMA is set to true, the transaction is completed by using DMA mode, when set to false the transaction is completed in Interrupt mode.

1.3.7 Using Callback Mode

By default, device drivers do not provide callbacks to the application (**NOTE**: Not all drivers provide callback mode support). If required, an application can register the callback with the driver after it is opened. The callbacks are not required for a typical application and not recommended to use, except for "event-driven" peripherals such as accelerometers. Refer to *Motivation for Avoiding Callbacks* to understand the rational for avoiding callbacks.

The following API is provided to register an optional callback and, thereby, to place the driver into Callback mode. When a callback is registered, peripheral errors and buffers are not returned with the `adi_xxx_GetBuffer` API call. The buffer pointer and peripheral errors are passed back to the application as callback arguments. If the application calls the `adi_xxx_GetBuffer` after registering the callback, the call returns an error. For more information, see *Peripheral Error Reporting*, which explains how errors are reported.

The API "un-registers" the callback if called with a NULL callback parameter.

adi_xxx_RegisterCallback

```
ADI_XXX_RESULT adi_xxx_RegisterCallback (
        ADI_XXX_HANDLE const hDevice,
        ADI_CALLBACK         pfCallback,
        void * const         pCBParam
 );
```

This API will place the driver into callback mode.

Callback Routines

All application callback routines are of type `ADI_CALLBACK`.

The definition of `ADI_CALLBACK` is as follows.

```
typedef void (* ADI_CALLBACK) ( /* Callback function pointer */
        void     pCBParam,      /* Client supplied callback param */
        uint32_t Event,         /* Event ID specific to the Driver/Service */
        void     pArg           /* Pointer to the event specific argument */
);
```

Callbacks are called by the driver when one of the following event types occurs:

1. A read or write transaction is complete.The argument `pArg` that is passed back is the address of the buffer. At this point the application owns the buffer.
2. An error has occurred during the read or write transaction. `pArg` contains the error code(s) for the driver.

Each driver documents the various "events" that can occur and cause a callback. It is the application's responsibility to process the event in the callback and take an appropriate action. If the cause of the callback is a *transaction complete* event, the application must synchronize the event with the application. Synchronization can be accomplished via a simple global variable or, in the context of an RTOS, via a semaphore.

Callbacks operate at interrupt level, so care must be taken to minimize the amount of code executed inside of the interrupt.

Callback mode is required for "event-driven" controllers.

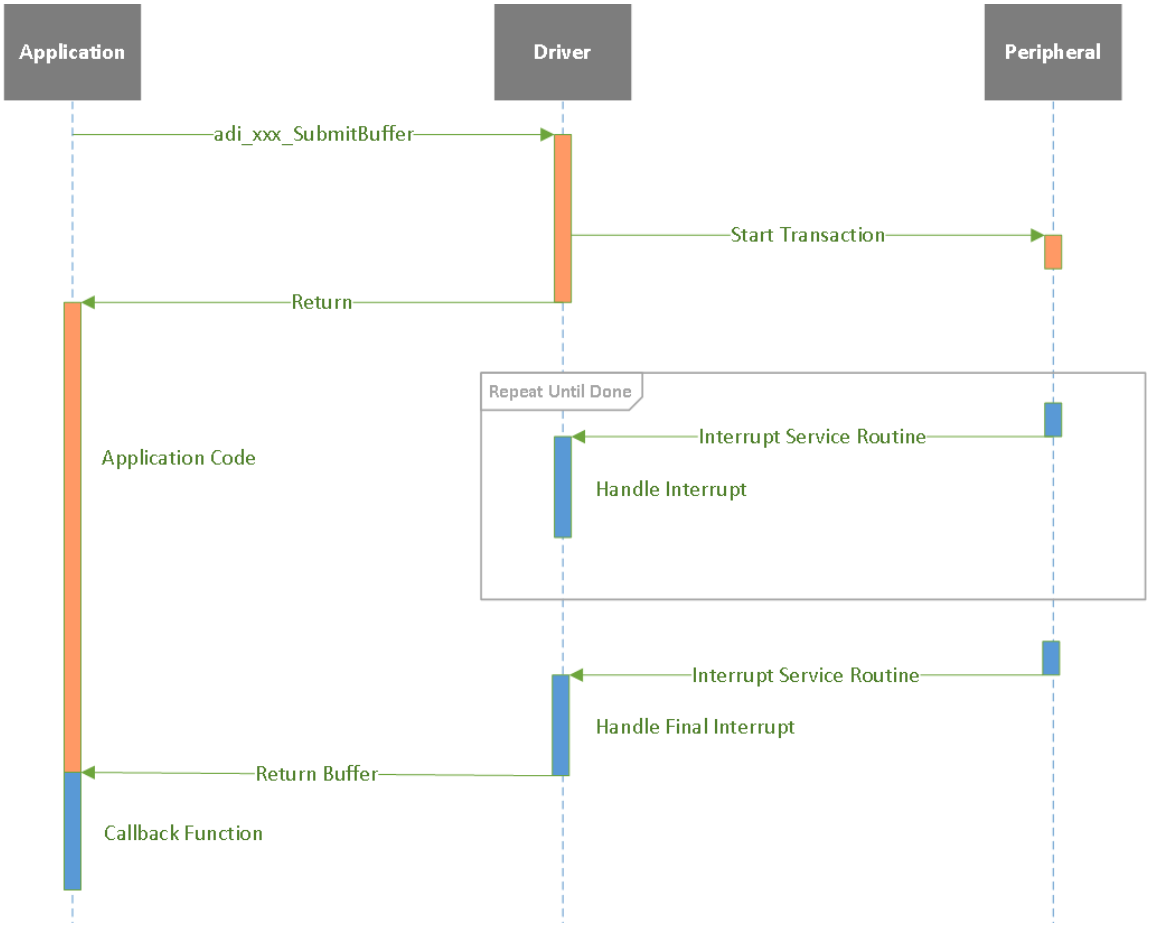
For controllers that are "event data driven", such as accelerometers, captouch, or touchscreen controllers, blocking mode reads and writes are supported only in conjunction with Callback mode.

Supporting non-blocking calls for these devices requires too much complexity in the driver, and the end result is a less efficient I/O for the application (that is, non-blocking requires more overhead than blocking, resulting in slower I/O). Non-blocking requires the underlying bus driver (I2C or SPI) to remain in an open state, which prevents any other context from using the bus driver.

For all of these reasons, the open API for these devices requires a callback.

The callback event indicates that data is ready to be read. The callback must synchronize this event with the application because the blocking read call cannot be made from within the callback (the callback is operating at interrupt level, and a blocking-mode read, while at interrupt level, results in erroneous behavior). The blocking-mode read call must be made at application level.

The following sequence diagram shows how the non-blocking mode APIs are used to interact with the device driver when using callback routines.



1.3.8 Peripheral Error Reporting

If a callback is registered, peripheral errors and DMA errors are reported via the callback. If a callback is not registered, peripheral errors and DMA errors are reported via the `adi_xxx_GetBuffer` or `adi_xxx_Read/Write` API calls. The `adi_xxx_GetBuffer` and `adi_xxx_Read/Write` calls will return a single error code (`ADI_XXX_HW_ERROR`) upon detecting an error. An application can examine the HwError (passed as pointer to `adi_xxx_GetBuffer` or `adi_xxx_Read/Write APIs)` to find out the exact cause of an error. The driver will logically OR all the errors that has occurred before calling the adi_xxx_GetBuffer API and clear them once they are reported.

Hardware error enumeration are defined such that the errors are logically ordered. For example:

```
typedef enum
 {
 ADI_XXX_NO_HW_ERR  = 0, /* No Errors were detected. */
 ADI_XXX_HW_ERR_OVF = 1, /* Overflow error was detected. */
 ADI_XXX_HW_ERR_UFL = 2, /* Underflow error was detected. */
 ADI_XXX_HW_ERR_DMA = 4, /* DMA error was detected. */
 ...
 }
```

1.4 Motivation for Avoiding Callbacks

There are a number of reason why applications should avoid callbacks.

- **Operating at Interrupt Level**

The callback is invoked from the Interrupt Service Routine. Therefore, the callback is operating at interrupt level. This gives the application supervisor mode capability. The application will have full access to the machine's MMRs and to the machine's privileged instructions.